WebAssembly Literacy

🙏 Retweet/Like pinned tweet on my 👉 twitter account as a token of appreciation

For great courses please subscribe to https://frontendmasters.com/

For better readability, set your browser zoom to 125% and reload the page.

If you like this kind of ebook, please also check the other's about visualization 💥 . Special thanks to my wife and children for the time lost while writing this, and to the awesome people at Mozilla (Alon Zakai, creator of Emscripten and co-creator of WebAssembly, is an ex-Mozillan), and to Wassim Chegham for the initial big retweet and kind words!

Wassim CHEGHAM

Are you looking for a great resource about #WebAssembly ? @ChambonMarc got you covered. Check out his FREE ebook, it's loaded with some great content, from Zero to Hero guarantee👌

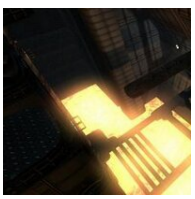
Karsten SCHMIDT

Whole book project is a great undertaking and a good reference for something i'm working on at the moment (wasm codegen for thi.ng/shader-ast)

Jay PHELPS

Very cool!!

Alon ZAKAI



Very nice!

0. Foreword

You want to learn WebAssembly (Wasm) for coding web apps in [replace by your favorite language] instead of Javascript and are not interested by the *WebAssembly Text Format* (WAT) : **THIS EBOOK IS NOT FOR YOU !**
Indeed, there're already many great tutorials about the amazing Emscripten project (C/C++) or about Rust or Go in order to enter new realms of web development without javascript and capitalizing on existing code bases.
Else you know javascript but not very well (use it just for jQuery plugins) : **THIS EBOOK IS VERY PROBABLY NOT FOR YOU !**

Else you love modern frontend technologies and practices, and use them to develop web apps offering to your users experiences similar to native ones :

- snappy UI;
- damn fast loading time;
- offline mode;
- nearly no waiting time or UI freezing during information processing.

Concretely, you're familiar with the actions below to reach these goals :

- writing modern CSS / SVG to manage as much animations as possible and hence offload the javascript thread;
- using tiny yet powerful frameworks (ex: Preat, Redux, StencilJs, Svelte…);
- harnessing the Cache and Service Worker APIs (look for Progressive Web Apps) for offline applications;
- following good javascript practices :
    - avoiding DOM accesses (relatively slow) when not absolutely necessary;
    - using async patterns (Promises, generators) properly;
    - not declaring new local variables in frequently called functions to avoid too much triggering of the garbage collector (=> temporary UI freezes);
    - using native ES6 methods when your application target modern browsers;
    - using Web Workers - to create additional threads - if some tasks can be executed in parallel;
    - using WebGL to speed up drawings, or heavy operations (scientific libraries, artificial intelligence...).

Still, your application has some bottlenecks. You've heard about the WebAssembly revolution, but the situation is not severe enough to rewrite the whole app from scratch in C++ or Rust - that you don't know. Indeed, your bundle size is not very big, and instead the problem lies in a dozen of slow functions. Moreover, you want to experiment with the Wasm technology without learning lots of tools. Above all, you want clear and succinct explanations but less dry than the official documentation. **THIS BOOK IS PROBABLY FOR YOU!** Indeed, each chapter show practical examples, that you can test in a few seconds by copying and pasting them into your browser console. Also, code samples are littered with detailed comments! For prerequisites, please check Appendix A.0.

The official documentation (formal language) and test suite (note: lots of syntactic sugar) are precious resources for those looking further, or to explore all the Wasm operators :

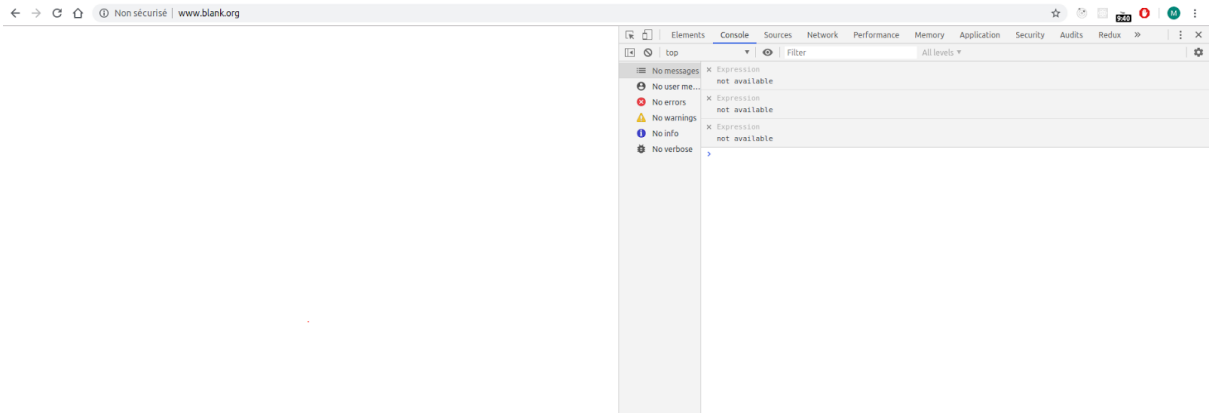
- http://webassembly.github.io/spec/core/binary/index.html
- https://github.com/WebAssembly/spec/tree/master/test/core

Instructions for efficient assimilation:

1. Please start reading from the beginning but for the Appendix (quite concise book relative to its scope);
2. Browse to www.blank.org or any website that allows direct Wasm code generation;
3. Open your browser console (typically: Ctrl + Shift + I or Cmd + Opt + I);
4. Copy-paste the code snippets when instructed so, usually at chapters named "Code summary"

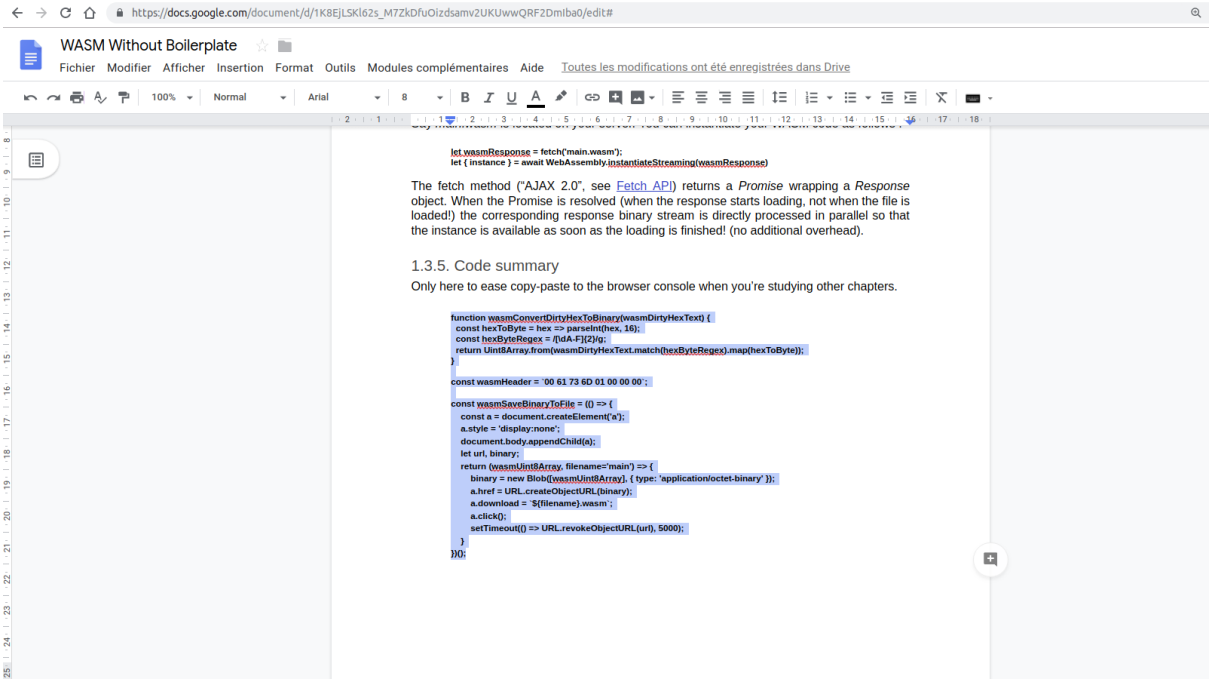
For those who are not the main public of this book, like IT students not familiar with the browser environment - here is the procedure detailed for chapter 1.4.2. :
In the end of of this chapter (page 19), you're are told to copy and paste two code snippets into your browser console to test/study the code example :

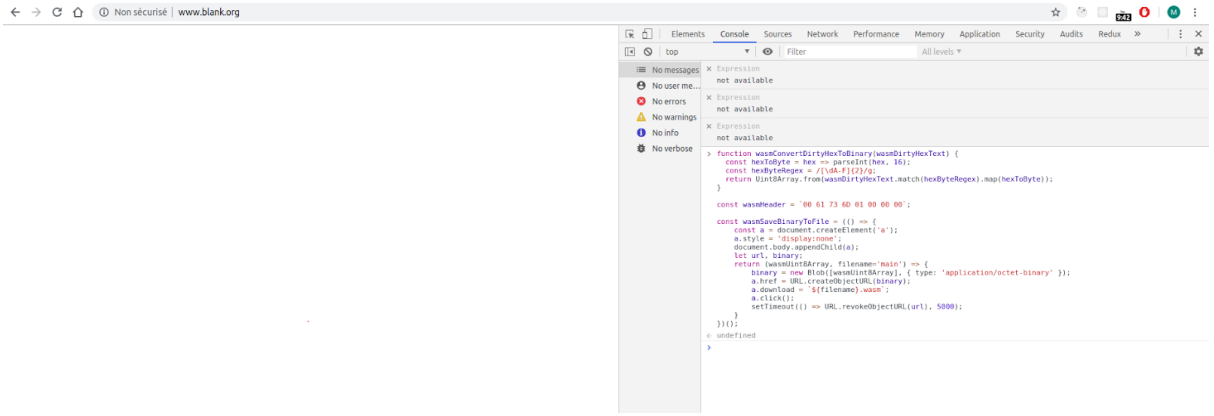
1. Open your navigator and browse to www.blank.org, then open the browser console



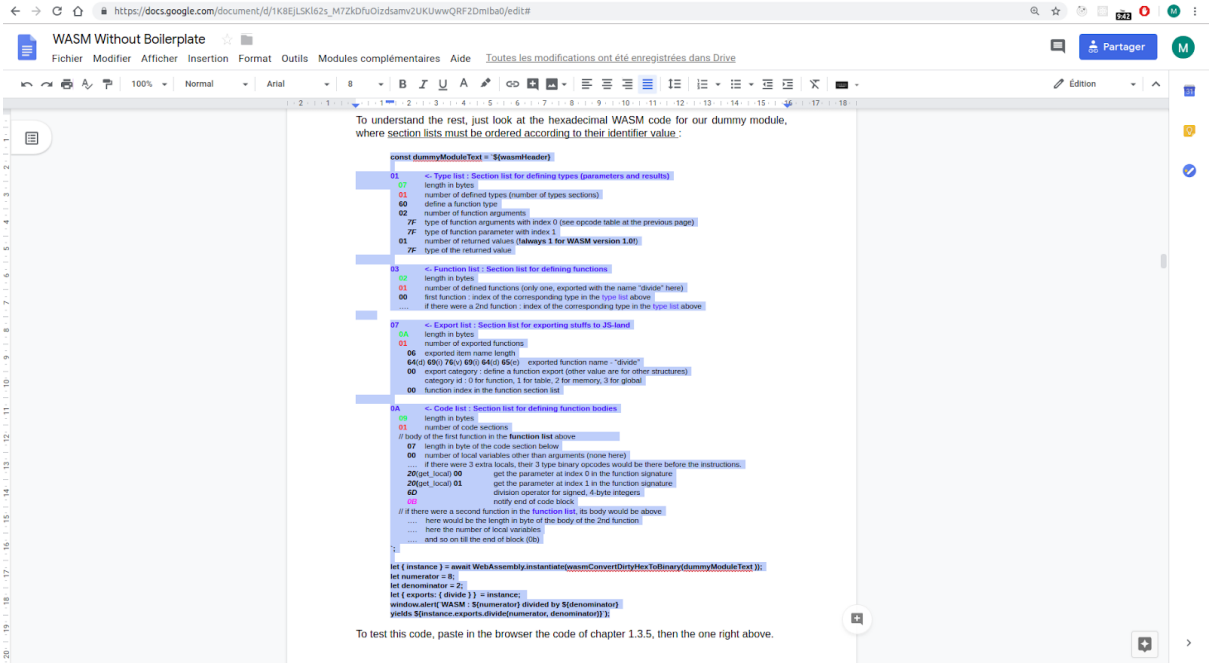
The console is the sub-window on the bottom-right with the ">" prompt.

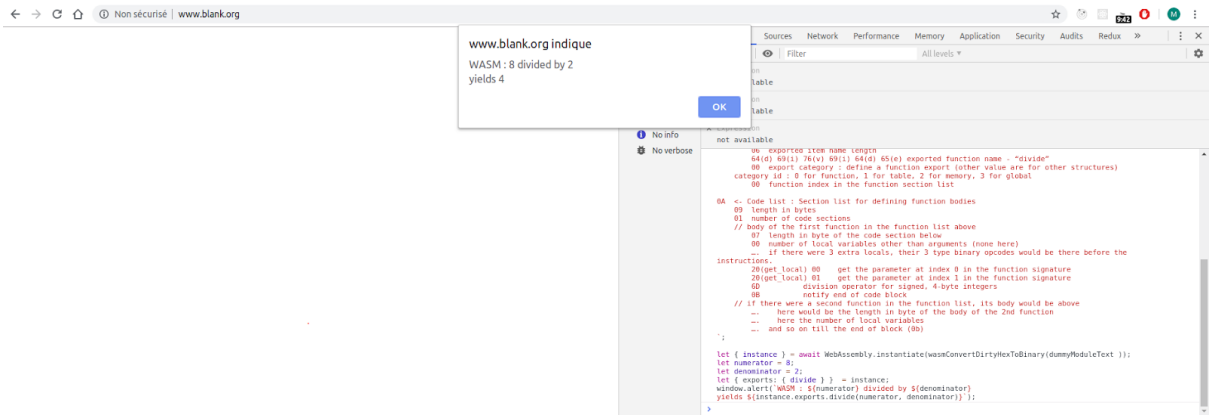
2. As told, copy the first code snipped from chapter 1.3.5, then copy it into the browser console then naturally type “enter” to run it.





3. As told, also repeat the procedure but for the second code snippet :





You can see the result of running the code on the last picture above : a dialog displaying the result of a calculation purely performed with Wasm code!

Note : if you want to tweak an example, refresh the browser (shortcut: Ctrl + R) to clear the variables in order to avoid these errors : *Uncaught SyntaxError: Identifier '.....' has already been declared .. then paste anew the code.*

Around the middle of the book, all the code is kept at one place (Wasm utilities repeated with the examples) to avoid your jumping around the chapters to copy code snippets.

1. Introduction

1.1. Context

As stated by Kyle Simpson, "*Javascript is as related to Java as Carnival is to Car*". Indeed, the name of the language wisely designed in a record time by Brendan Eich was chosen only for marketing purposes, at its birth at Netscape :

- Javascript is not derived from Java (functional aspects, based on prototypes);
- Javascript is not a scripting language anymore.

The second point is extremely important to understand Wasm (WebAssembly) : Javascript has become very fast since its inception, and on modern browsers, javascript code is compiled Just-In-Time (JIT) by their javascript engines (SpiderMonkey, V8, etc) using state-of-the-art technologies :

1. The first time a javascript code snippet is loaded and executed, many intermediate and sequential steps slow down the process, above all the parsing steps, i.e. the traduction of the textual source code into intermediate compilation structures.
2. Also, the first time a part of a code such as a function is executed, it's also quite slow because - as javascript is not typed - the javascript engine must infer the variable types at runtime and find the most efficient way - through iterative introspection - to execute the code : it transforms the intermediate compilation structures into final ones, then machine code (depending on the platform, x86 or ARM...), and executes it. Then, according to the runtime context (e.g. are the value types constant?) and to the recorded performance, the JIT compiler tweaks then rebuilds the final structures to improve the global performance for the next rounds.
3. Once these "warming" steps are finished, and if the function keeps running - say in a loop - with similar argument types, the final compilation structures are stable so the javascript engine can achieve nearly native performance (provided the javascript code is written in a way that avoids the garbage collector triggering).

Conclusion : if one can directly produce final compilation structures with typed values, one can short-circuit steps 1 and 2 to go directly to step 3 where performance is a nearly native one (only 20-50% slower than machine code, incredible when you think about the complexity of a browser.. ..engineers working on browsers are unknown heroes!). This is basically what Wasm does : representing final compilation structures by an **unambiguous and generic far lower level code**, so that the final step doesn't have to be very smart (= CPU intensive) to convert it extremely quickly to machine code.

1.2. Fundamentals

What is Wasm more concretely ?

1.2.1. Isolated runtimes via modules

Wasm code is only executed inside a highly-secured - isolated - runtime environnement, not specific to the web in spite of the name (see WASI). There's no main application nor library. Instead, Wasm binary code lives inside autonomous *modules*. A Wasm *module* cannot access the operating system nor javascript memory and functions, but if they have been explicitly imported.

Each *module* corresponds to a binary file - by convention with the *.wasm* extension.
A ready-to-run *module* is called an *instance.* A *module* is organized into *sections*.
There're 11 useful section categories (only the 4 first ones are required for the introduction) :

- *type sections* defining function signatures (result and argument numbers and types);
- *code sections* defining function bodies (= where the statements can be found);
- *function sections* declaring functions via references to their signatures (and bodies in binary code);
- *export sections* giving the javascript code access to Wasm functions or values;
- *global sections* sharing variables between Wasm functions;
- *one memory section* acting as runtime memory (*heap*) when one processes as example huge structures (this also allows to work on javascript typed arrays);
- *one table section* storing references to functions for indirect function calls (like pointers);
- *one start section* notifying which function to start when the module is instantiated - to simulate a "main" function);
- *import sections* giving the module access to host functions (ex: DOM manipulation), constant values or a table - as defined in the table section - from other modules;
- *data* and *element sections (only one of each)*, to initialize the *memory* and *table sections*, respectively.

Hence, a *instance* communicates with the host - the browser in this book - and with other *instances* through shared structures. In the browser, this allows multithreading for an even higher performance by letting multiple *web workers* - threads in the browser - manage their own *instances (chapter 4)*.

Notes :

- *Memory and table sections are limited to one per module for Wasm 1.0, but this constraint will be probably removed in future versions;*
- *There's a 12th section called custom section, to add metadata to a module (e.g. name).*

1.2.2. Svelte virtual machine

As said at the beginning, the JIT compiler last steps perform nearly as fast as machine code. Hence, it's wiser to target these steps than directly writing machine code, by packing them into a *virtual machine*. Each module is thus executed by such a *virtual machine*.

Moreover, this *virtual machine* relies on a pseudo *stack* architecture. A pseudo one as it's just an interface chosen for its simplicity, and doesn't reflect the actual implementation.

For self-taught web developers, the principle of a *stack* machine is as follows : a *unique stack* - basically like a regular javascript array but restricted to the **push** and **pop** operations - is the only mediator between the *virtual machine* instructions or functions and the values they operate on and return. The principles are easily understood with the basic example below :

Say that you want to write a dummy "divide" function (ex: *divide(8, 2) = 4*) :

1) At the beginning - *Stack* : ***[ ]***
2) You must explicitly push the arguments upon the stack. This is done in Wasm through "get" operators (e.g. **get_local**), followed by an index :
   After **get_local 00** (loading 1st argument) - *Stack : **[8]***
   After **get_local 01** (loading 2nd argument) - *Stack : **[8, 2]***
   If instead of using the second argument you want to provide your own value, you must use a "const" operator (like **i32.const**) to add 2 upon the stack : **i32.const 2**
3) The stack can be used by a Wasm operator (e.g. **i32.div_s** for signed division):
   The operator you use at this step determines how many values will be popped and processed (e.g. 2 values for a binary operator like **i32.div_s**, or more..):
   When **i32.div_s** starts (popping 2 values from the stack) - Stack : ***[ ]***
   i32.div_s with 8 and 2 yields 4.
   Hence, when **i32.div_s** finishes, 4 is pushed upon the stack) - Stack : ***[4]***
4) In Wasm functions, values are implicitly returned : the *return value* operator doesn't exist (there's a **return** operator but only to exit a code block, see chapter 3.4.1., page 47) as the value returned by a Wasm function is simply the last value at the top of stack. Hence, the body of a Wasm function simply ends after the last operation.

Note : The maximum size of this stack is not defined in Wasm 1.0, so it mays differ between browser implementations, but it is naturally big enough so that you can forget about it when writing useful programs.

1.2.3. Value representations

Note: you don't need to understand everything in details for basic understanding of Wasm code.

Values are naturally typed, with only 4 ones right now : 4-byte integer (**i32**) and floating point (**f32**) types, and 8-byte integer (**f64**) and floating point (**i64**) types. By the way numerical values in javascript are **f64** (when you use them inside functions), even for integers!

Integers are either signed or unsigned, and the *two's complement* convention is used for signed integers : to express a negative value, one takes the binary representation of the positive value, inverts every bit then adds 1 (ex: 3 = **0000...0011** => **1111...1100** => **1111...1101**).

Floating point values (floats) rely on the IEEE-754-2008 encoding (see Appendix A.1.).

The *virtual machine* is ALWAYS *little-endian* (known exception: IEEE-754 bytes). This means that the byte order for data processing is right-to-left, the unnatural one (small byte values read before the big ones - this is also the opposite of the network endianness, *big-endian*). Hence a 8-byte value like - in hexadecimal - **12 34 56 89** is represented by **89 56 34 12**.

Internal values are stored using signed LEB128 compression (see Appendix A.2.).
Nothing changes for binary encoding if a value is between -64 and 63, included (the encoded value is as the default one).

Characters rely on the UTF-8 encoding, retro-compatible with the ASCII's for latin characters and symbols (see Appendix A.3.). A character sequence (string) used in Wasm doesn't end with a null byte as in C/C++ : instead, strings inside Wasm code are prepended with their byte length. As a Wasm module only processes binary data, the **TextDecoder** constructor must be used to decode a **Uint8Array** representing a character sequences (**binaryText**), like this :

```
const hexToByte = hex => parseInt(hex, 16);                  // Convert hexadecimal string to values.
const utf8String = ['D7', 'A9', 'D7', '9C', 'D7', '95', 'D7', '9D'].map(hexToByte);  // always left-to-right!
const binaryText = Uint8Array.from(utf8String);
const string = new TextDecoder().decode(binaryText);      // TextDecoder decode by default in UTF-8
console.log(string);              // right-to-left display is automatic in the console  (inferred from UTF-8)
```

Paste this code snippet into your browser console to test the process.

As a quick reference, you can find at the penultimate page of this book some useful **hexadecimal** UTF-8 correspondences to latin characters.

1.2.4. Concise and well-structured languages (binary and WAT)

A Wasm binary is a compact, well-structured (= unambiguous) format - consequently offering ultra-fast, single pass compilation of the code. There's also a more convenient, textual representation of the Wasm binary format, called the *WebAssembly Text format* (WAT), consisting of *S-expressions* and syntactic sugar (see chapter 1.4.3.). A *S-expression* is a very basic textual tree representation of the code structure using mere parenthesis. Say you have a structure **A** containing sub-structures **A1** and **A2** - called *nodes* - then its *S-expression* is :

```
(A (A1) (A2))
```

Or reformatted as follows for better readability :

```
(A
        (A1)
        (A2)
)
```

Moreover, if **A1** also contains **A11** and **A12** *nodes*, then its *S-expression* is :

```
(A
        (A1
                (A11)
                (A12)
        )
        (A2)
)
```

In Wasm, the top node (**A**) is called the *module* and first children (**A1**, **A2**) are *sections*.

1.3. Concrete Wasm basics

Being faithful to this book title, let's build Wasm modules without boilerplate. To ease our task, one starts with a dummy, i.e. empty module.

The function below converts a text - **wasmDirtyHexText** - containing a hexadecimal byte sequence (e.g: **FF**) representing a *module* - into binary code :

```
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  const hexToByte = hex => parseInt(hex, 16);
  // This regular expression only extracts uppercase'd hexadecimal byte notations
  const hexByteRegex = /[\dA-F]{2}/g;
  // Extract an array of hexadecimal bytes from the text, then convert it (map) to bytes.
  const wasmValueArray = wasmDirtyHexText.match(hexByteRegex).map(hexToByte);
  // Cast values into unsigned bytes
  const wasmBinaryArray = Uint8Array.from(wasmValueArray);
  // Return wasm data
  return wasmBinaryArray;   // code binary is indifferently of types Uint8Array or Uint8Array.buffer (ArrayBuffer)
}
```

Note: The regular expression only extracts the hexadecimal bytes with UPPERCASE letters (e.g: **0A***), allowing to litter* **wasmDirtyHexText** *with lowercase'd comments and single digits.*

1.3.1. Minimum module

In order to indicate that a code binary is a Wasm module, the former must be prepended with a 8-byte “magic value” header (**wasmHeader** in “dirty” (with comments) hexadecimal text) :

```
const wasmHeader = `
00(null) 61(a) 73(s) 6D(m)      header 1/2 : ascii "asm" inside 4 bits =  Wasm module
01 00 00 00`;                   header 2/2 : WebAssembly version number (1.0)
```

The minimum Wasm module is just this header, and is executed as follows :

```
let emptyWasmModule = wasmConvertDirtyHexToBinary(wasmHeader);
let { instance } = await WebAssembly.instantiate(emptyWasmModule);        // get instance inside the module
```

WebAssembly.instantiate compiles then instantiates the Wasm binary code (**Uint8Array** or **ArrayBuffer**) and returns a Promise wrapping the module having an **instance** (chapter 1.2.1).

Test for yourself! Open the browser console, and paste the 3 code snippets in this order :

- the **wasmConvertDirtyHexToBinary** function (previous page);
- the **wasmHeader** variable;
- The 2 lines just above.

Even if nothing apparently happens, a Wasm module has just come to life inside your page !

The WebAssembly Text (WAT) code associated with this empty module is just :

```
(module)          ;; you can leave line comments inside WAT code using ";;".
```

1.3.2. How to save Wasm modules

You can simply use the function below to save to your computer the Wasm binary produced by the **wasmDirtyHexToBinary** function (chapter 1.3), for future use on a server:

```
// Immediately Invoked Function Execution (IIFE) to create the inner function environment  and return this function
const wasmSaveBinaryToFile = (() => {
   // Closure to keep alive the function environnement between each run
   // (the garbage collector doesn't remove them because their references are kept here)
   const a = document.createElement('a');
   a.style = 'display:none';
   document.body.appendChild(a);
   let url, binary;
   return (wasmUint8Array, filename='main') => {
      binary = new Blob([wasmUint8Array], { type: 'application/octet-binary' });
      a.href = URL.createObjectURL(binary);
      a.download = `${filename}.wasm`;
      a.click();
      // 5 seconds should be sufficient to save the file before destroying the URL object.
      setTimeout(() => URL.revokeObjectURL(url), 5000);
   }
})();
```

Test it by pasting the code of this function into the browser console - provided you have followed all the previous steps of this chapter - then also paste the code below :

```
wasmSaveBinaryToFile(wasmConvertDirtyHexToBinary(wasmHeader));
```

1.3.3. Module generation from WAT

WebAssembly Studio is a great free service (kudos to its authors!) to test and generate Wasm code written as WAT. To generate the same module :

1) Go to this website and choose *Empty Wat Project* in the *Create New Project* dialog :
   https://webassembly.studio/
2) Click on main.wat in the navigation window on the left, and replace content with :
   (module)
3) Build and run the project (CtrlCmd + Maj + Enter)
4) Download the project folder and get *main.wasm* file from the *out* sub-folder.

Note : WebAssembly Studio and many similar services currently do not work for bleeding-edge features. As an example for shared memory, required for multi-processing, it throws this error: "memories may not be shared". See appendix A.4. for an alternative.

1.3.4. How to efficiently load Wasm modules

Say *main.wasm* is located on your server. You can instantiate your Wasm code as follows :

```
let wasmResponse = fetch('main.wasm');
let { instance } = await WebAssembly.instantiateStreaming(wasmResponse)
```

The fetch method ("AJAX 2.0", see Fetch API) returns a *Promise* wrapping a *Response* object. When the Promise is resolved (when the response starts loading, not when the file is loaded!) the corresponding response Wasm binary stream is directly compiled and instantiated *in parallel* so that **instance** is available as soon as the loading is finished!

1.3.5. Code summary

Only here to ease copy-paste to the browser console when you're studying other chapters.

```
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  const hexToByte = hex => parseInt(hex, 16);
  const hexByteRegex = /[\dA-F]{2}/g;
  return Uint8Array.from(wasmDirtyHexText.match(hexByteRegex).map(hexToByte));
}

const wasmHeader = `00 61 73 6D 01 00 00 00`;

const wasmSaveBinaryToFile = (() => {
    const a = document.createElement('a');
    a.style = 'display:none';
    document.body.appendChild(a);
    let url, binary;
    return (wasmUint8Array, filename='main') => {
        binary = new Blob([wasmUint8Array], { type: 'application/octet-binary' });
        a.href = URL.createObjectURL(binary);
        a.download = `${filename}.wasm`;
        a.click();
        setTimeout(() => URL.revokeObjectURL(url), 5000);
    }
})();
```

1.4. Minimum useful Wasm module

This concrete chapter will help you digest the concepts by coding a dummy “divide” function performing an integer division. Let’s dive slightly more deeply into a module structure before.

1.4.1. Module interface and structure - 101

Wasm items (function, etc) must be exported to be useful. The WAT correspondence is :

```
(module
        ....                            ;; other required sections
        (export "item0"                 ;; name of the first exported object in JS-land
                ....                    ;; other stuffs
        )
        (export "item1"                 ;; name of the second exported object.
                ....                    ;; other stuffs
        )
)
```

In JS-land, you can find these functions inside the **exports** field of the **instance** :

```
let { instance } = instantiateStreaming('main.wasm');
const whatever = instance.exports.item0;        // for a generic export
instance.exports.item1(/* arguments here */);   // if item1 is a function
```

As **export** sections are <u>not specific to functions</u> (you can export other types of items), you must specify the nature of each one with a child node (e.g. **func**).

```
(module
        ....                            ;; other required sections
        (export "item1"
                (func                   ;; child node : item1 is a function
                        ....            ;; other required stuffs
                )
        )
)
```

The **func** keyword above says that, and its internal node (number **1** below) is a <u>reference via a top-to-bottom index</u> to an eponymous (= same name) module section, a *function* section :

```
(module
        ( func                          ;; function section with index 0
                ....                    ;; other required things
        )
        (func                           ;; function section with index 1 : function labelled as “divide”
                ....                    ;; other required things
        )
        (export "divide"
                (func   1)              ;; index of the exported function in the module
        )
)
```

Each section category has its own index system, so nothing changes if we have instead:

```
(module
        ( func                          ;; function section with index 0 in the function section list
                ....                    ;; other required things
        )
        ( .... )                        ;; section with category different than function (func)
        (func                           ;; this <u>function section still has index 1</u> -
                                        ;; - because indexed in the function section list only
                ....                    ;; other required things
        )
)
```

Functions must be signed - to defined the number and types of their arguments and result - thanks to a **type** section, whose logic is similar (child with references) :

```
(module
        (type                                   ;; type section with index 0 in the type section list
                ....                            ;; other required things
        )
        (func                                   ;; function with index 0 ("divide") in the function section list
                (type     0)                    ;; index of the type (function signature)  in the module
                ....                            ;; other required things
        )
        (export "divide"
                (func     0)                    ;; exported function referenced by an index
        )
)
```

A *type section* must contain a node specifying the nature of the type (**func** for a function), itself containing nodes to define the parameters (**param**) and results (**result**) :

```
(module
        (type
                (func                           ;; type category
                        (param                  ;; node with the ordered argument types
                                i32             ;; child node: argument 1 is a 4-byte integer
                                i32             ;; child node: argument 2 ...
                        )
                        (result                 ;; node with the result type(s)
                                i32
                        )
                )
        )
        (func
                (type     0)
                ....                            ;; other required things
        )
        (export "divide"
                (func     0)
        )
)
```

If we need extra local variables aside the parameters - not the case with our "divide" function - a **local** node must be added right after the **type** node in the **func** parent. Finally, the body function starts right after (see chapter 1.2.2) :

```
(module
        (type
                (func
                        (param
                                i32
                                i32
                        )
                        (result   i32)
                )
        )
        (func
                (type     0)
                (local f32 i64 f64)             ;; 3 local variables of various types in addition to the arguments.
                get_local 0                     ;; push argument 1 upon the stack
                get_local 1                     ;; push argument 2 upon the stack
                i32.div_s                       ;; pop 2 values from  the stack, perform integer division…
                                                ;; … and push the result upon the stack.
        )
        (export "divide"
                (func     0)
        )
)
```

Both **param**s (2 ones here) and **local**s (3 ones) must be pushed upon the *stack* of the *virtual machine* before calling operators or functions, like this with the **get_local** instruction :

- **get_local 00** : push the 1st parameter (**i32** numerator for “divide”);
- **get_local 01** : push the 2nd parameter (**i32** denominator for “divide”);
- **get_local 02** : push the **f32** value;
- **get_local 04** : push the **f64** value.

1.4.2. Wasm binary composition

If we don’t use WAT syntactic sugar, the structural correspondence between WAT and binary codes is nearly one-to-one.

For binary Wasm, sections of a same category C (C = *type* or *export*, etc) are grouped into a single section list (in the order they appear) :

C section list is equivalent to : [(C section with index 0), (C section with index 1) …]

Hence, all type sections will be grouped into a single type section list in the binary code, a first byte group defining the 1st section (e.g. type section with index 0), directly followed by a second byte group defining the 2nd section (e.g. type section with index 1), etc.

Each section list is prepended with a *byte identifier* corresponding to its section category. Such a Wasm identifier has an integer value in the 0-11 range :

- **01** for the *type* (**type**) section list;
- **03** for the *function* (**func**) section list;
- **07** for *expor*t (**export**) section list;
- **0A** for the *code* (function body) section list (there’s no code section in WAT as a function body can only be nested inside a function section);
- more to come in the next chapters.

Also, right after each of these identifiers and before the binary sections there’re :

- a value defining the length in bytes of the section list;
- another value specifying the number of sections in the list.

For a binary *code section*, the byte sequence representing a function body is simply the binary version of the values, types and WAT instructions, defined by *tokens*.
As an example for our “divide” function:

WAT token	**f64** (value type)	**f32** (value type)	**i64** (value type)	**i32** (value type)	**end** (of a code block)	**get_local**	**i32.div_s**
Binary token	***7C***	***7D***	***7E***	***7F***	***0B***	***20***	***6D***

WAT instructions are called *opcodes.* For the full list check :
http://webassembly.github.io/spec/core/appendix/index-instructions.html

To understand the rest, just look at the hexadecimal Wasm code for our dummy module, where section lists must be ordered according to their identifier value :

```
const dummyModuleText = `${wasmHeader}

01          <- Type list : Section list for defining types (parameters and results)
  07        length in bytes
  01        number of defined types (number of types sections)
  60        define a function type
  02        number of function arguments
    7F      type of function arguments with index 0 (see opcode table at the previous page)
    7F      type of function parameter with index 1
  01        number of returned values (!always 1 for Wasm version 1.0!)
    7F      type of the returned value

03          <- Function list : Section list for defining functions
  02        length in bytes
  01        number of defined functions (only one, exported with the name “divide” here)
  00        first function : index of the corresponding type in the type list above
  ....      if there were a 2nd function : index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
  0A        length in bytes
  01        number of exported functions
    06      exported item name length
    64(d) 69(i) 76(v) 69(i) 64(d) 65(e)    exported function name - “divide”
    00      export category : define a function export (other value are for other structures)
            category id : 0 for function, 1 for table, 2 for memory, 3 for global
    00      function index in the function section list

0A          <- Code list : Section list for defining function bodies
  09        length in bytes
  01        number of code sections
  // body of the first function in the function list above
    07      length in byte of the code section below
    00      number of local variables other than arguments (none here)
    ....    if there were 3 extra locals, their 3 type binary opcodes would be there before the instructions.
    20(get_local) 00          get the parameter at index 0 in the function signature
    20(get_local) 01          get the parameter at index 1 in the function signature
    6D                        division operator for signed, 4-byte integers
    0B                        notify end of code block
  // if there were a second function in the function list, its body would be above
      ....  here would be the length in byte of the body of the 2nd function
      ....  here the number of local variables
      ....  and so on till the end of block (0b)
`;

let { instance } = await WebAssembly.instantiate(wasmConvertDirtyHexToBinary(dummyModuleText ));
let numerator = 8;
let denominator = 2;
let { exports: { divide } }  = instance;
window.alert(`Wasm : ${numerator} divided by ${denominator}
yields ${instance.exports.divide(numerator, denominator)}`);
```

To test this code, paste in the browser the code of chapter 1.3.5, then the one right above.

1.4.3. Syntactic sugar

Code sections are not used inside WAT code. Instead, the function body is nested in its related *function section*. The valid code is thus (less formatted for the sake of the space) :

```
(module
	(type
		(func (param i32 i32) (result  i32))
	)
	(func (type 0)
		get_local 0
		get_local 1
		i32.div_s
	)
	(export "divide" (func 0))
)
```

However, a given *type* (function signature) is usually directly “spread” inside a *function section* instead of being referenced, like this :

```
(module
	(func (param i32 i32)  (result i32)
		get_local 0
		get_local 1
		i32.div_s
	)
	(export "divide" (func 0))
)
```

Also, **param**s (and also **local**s) can be splitted like this :

```
(module
	(func (param i32) (param i32)  (result i32)
		get_local 0
		get_local 1
		i32.div_s
	)
	(export "divide" (func 0))
)
```

To avoid mistakes while handling indices, you are encouraged to use named alias instead, which must be prepended with **$** and used as follows :

```
(module
	(func $divide (param $numerator i32) (param $denominator i32)  (result i32)
		get_local $numerator
		get_local $denominator
		i32.div_s
	)
	(export "divide" (func $divide))
)
```

You can also directly export a function within its section like this :

```
(module
	(func (export "divide")  (param $numerator i32) (param $denominator i32)  (result i32)
		get_local $numerator
		get_local $denominator
		i32.div_s
	)
)
```

Finally, you can also use a more familiar notation instead of the stack one :

```
(module
    (func (export "divide")  (param $numerator i32) (param $denominator i32)  (result i32)
        (i32.div_s
            (get_local $numerator)          ;; first “argument” for divide operator
            (get_local $denominator)        ;; second “argument” for divide operator
        )
    )
)
```

2. Practical integration

You should have a pretty clear understanding by now of a Wasm module. However, because of its isolation, a module like the previous one is quite limited as long as we cannot exchange more information with the browser. The goal if this chapter is to explore a sea of possibilities by adding more section categories (see chapter 1.4.2), whether directly in the Wasm code or from javascript.

2.1. Sharing values inside a module via globals

Global sections allow to declare variables global to a module (= shared between its functions). Such sections have the identifier with value 06. For better understanding, let’s modify our “divide” function as follows :

- name changed to “divideAndAdd”;
- one global variable added to the module, with initial value equals 16;
- “divideAndAdd” adds the global value to the result of the previous division.

2.1.1. WAT code

If you’ve read the first chapter, the WAT code of the tweaked function should be nearly self-explanatory :

```
(module
    (global                 ;; global section with index 0 in the global section list
        (mut i32)           ;; mut = mutable; replace (mut i32) by i32 if constant
        (i32.const 16)      ;; initial value of the global variable, passed with a i32.const operator
    )
    (type
        (func (param i32 i32) (result  i32))
    )
    (func (type 0)
        get_local 0
        get_local 1
        i32.div_s           ;; result of the division is pushed upon the stack
        get_global 0        ;; global referenced by its index
        i32.add             ;; pop both the division result and the global, and add them
    )
    (export "divideAndAdd" (func 0))
)
```

The only subtlety concerns the mutability of the global variable :

- if constant, the first child of the related global section is simply its type - **i32**;
- otherwise, the first child is also a new node containing **mut** followed by the type.

2.1.2. Binary code

WAT token	i32	end	get_local	i32.div32_s	i32.const	get_global	i32.add
Binary token	*7F*	*0B*	*20*	*6D*	*41*	*23*	*6A*

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;

const wasmGlobalExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   07(length in bytes) 01(number of defined types (number of types sections))
   60       define a function type
   02       number of function arguments
      7F    type of function arguments with index 0 (see opcode table at the previous page)
      7F    type of function parameter with index 1
   01       number of returned values (!always 1 for Wasm version 1.0!)
      7F    type of the returned value

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions (only one, exported with the name “divide” here))
   00       first function : index of the corresponding type in the type list above

06          <- Global list : Section list for defining global variables
   06(length in bytes) 01(number of defined global variables)
      7F    type of the first global
      01    mutability of the first variable (0 = constant, 1 = variable)
      41    10          code block to initialize the first global (pushing sixteen upon the stack)
      0B    notify end of code block
   // if there were a second global variable
      ....  here would be its type
      ....  here its mutability
      ....  here its code block for initialization, ending with 0b

07          <- Export list : Section list for exporting stuffs to JS-land
   10(length in bytes) 01(number of exported functions)
      0C    exported item name length
      64(d) 69(i) 76(v) 69(i) 64(d) 65(e) 41(A) 6E(n) 64(d) 41(A) 64(d) 64(d)          “divideAndAdd”
      00    export category : define a function export (category id : 0 for function, 1 for table, 2 for memory, 3 for
global)
      00    function index  in the function list above

0A          <- Code list : Section list for defining function bodies
   0C(length in bytes) 01(number of code sections)
   // body of the first function in the function list above
      0A    length in byte of the code section below
      00    number of extra local variables (none here)
      20(get_local) 00          get the parameter at index 0 in the function signature
      20(get_local) 01          get the parameter at index 1 in the function signature
      6D                        division operator for signed, 4-byte integers
      23(get_global) 00         get the global variable at index 0 in the global section list
      6A                        add operator for 4-byte integers
      0B                        notify end of code block
`;
const { instance } = await WebAssembly.instantiate(wasmConvertDirtyHexToBinary(wasmGlobalExample));
const numerator = 8;
const denominator = 2;
const { exports: { divideAndAdd } }  = instance;
window.alert(`Wasm - import : ${numerator} divided by ${denominator} plus 16
yields ${instance.exports.divideAndAdd(numerator, denominator)}`);
})();
```

Test this by pasting the code above in the browser console.

2.2. Interactions with Javascript via imports

Instead of setting a global value inside the WAT or binary code, one can choose its value at runtime, during the instance creation, through importation. *Import sections* have the identifier with value **02**.

2.2.1. Principle explanation

Except for functions, values cannot be directly imported : they must first be converted into Wasm objects through a **WebAssembly** constructor accepting a value (e.g. **globalValue**) and additional settings (e.g. **globalDescriptor**) :

```
let globalValue = 16;          // what you're interested to import
let globalDescriptor = {
   value: 'i32',               // type in Wasm-land
   mutable: false              // always false for global import (mutability forbidden at the moment for such imports)
};
let addedValue = new WebAssembly.Global(globalDescriptor, globalValue);
```

Also, a module imports a single object, and only during its instantiation (second parameter of the **instantiate** or **instantiateStreaming** methods). Moreover, it must be a 2-level nested structure, the first level acting as a namespace (e.g. **js** and **api** fields) for the imported items (e.g. **addedValue** and **login**). Hence, the javascript code is as follows :

```
let js = { addedValue };
let api = { login };           // provided login  is a function or a Wasm object
let wasmImport = { js, api };
let { instance } = await WebAssembly.instantiate(wasmBinary, wasmImport);
```

Let's resume the code of the previous chapter, but by importing a global constant value (e.g. **addedValue**) instead of creating it in the Wasm code.

2.2.1.1. WAT code

The WAT code mirrors the one from the chapter 2.1.2, with the following differences :

- explicit **global** section deleted;
- **import** section added, one for each item imported from JS-land.

```
(module
        (import                         ;; import section for a global named "addedValue" in JS-land
                "js"    "addedValue"    ;; import addedValue nested in the js field of wasmImport
                (global i32)            ;; import a constant 4-byte integer
        )                               ;; create an implicit first global section, with index 0.
        (type
                (func (param i32 i32) (result i32))
        )
        (func (type 0)
                get_local 0
                get_local 1
                i32.div_s
                get_global 0        ;; global referenced by its section index
                i32.add             ;; pop both the division result and the global, and add them
        )
        (export "divideAndAdd2" (func 0))
)
```

2.2.1.2. Binary code

Wasm token : **get_global** equivalence is **23** in binary

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmGlobalExample2 = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   07(length in bytes) 01(number of defined types (number of types sections))
   60       define a function type
   02       7F 7F       number and types of function arguments
   01       7F          number of returned values (!always 1 for Wasm version 1.0!)

02          <- Import list : Section list for importing items from javascript
   12(length in bytes) 01(number of import (only one, exported with the name “divide” here))
   02       import “namespace” length in byte.
   6A(j) 73(s)          import “namespace” : “js”
   0A       import name length in bytes
   61(a) 64(d) 64(d) 65(e) 64(d) 56(V) 61(a) 6C(l) 75(u) 65(e)              imported variable: “addedValue”
   03       import category : define a global import (category id : 0 for function, 1 for table, 2 for memory, 3 for global)
   7F       imported global value type (4-byte integer)
   00       imported global mutability (zero = constant, one = mutable)
   // if there were a second imported item
      ….    here would be respectively the namespace preceded by its length, the item name preceded by its length,
      ….    here would be the imported item category, type and mutability

03          <- Function list : Section list for defining functions
   02       length in bytes
   01       number of defined functions (only one, exported with the name “divide” here)
   00       first function : index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   11(length in bytes) 01(number of exported functions)
      0D    exported item name length
      64(d) 69(i) 76(v) 69(i) 64(d) 65(e) 41(A) 6E(n) 64(d) 41(A) 64(d) 64(d) 32(2) “divideAndAdd2”
      00    export category : define a function export (category id : 0 for function, 1 for table, 2 for memory, 3 for
global)
      00    function index in the function list above
0A          <- Code list : Section list for defining function bodies
   0C(length in bytes) 01(number of code sections)
      0A    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      20(get_local) 00          get the parameter at index 0 in the function signature
      20(get_local) 01          get the parameter at index 1 in the function signature
      6D                        division operator for signed, 4-byte integers
      23(get_global) 00         get the global variable at index 0 in the global section list (or implicit if imported)
      6A                        add operator for 4-byte integers
      0B                        notify end of code block
`;
const globalDescriptor = { value: 'i32', mutable: false };
const globalValue = 16;
const addedValue = new WebAssembly.Global(globalDescriptor, globalValue);
const wasmImport = { js: { addedValue } };
const { instance } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmGlobalExample2),
   wasmImport
);
const numerator = 8;
const denominator = 2;
const{ exports: { divideAndAdd2 } }  = instance;
window.alert(`Wasm - import : ${numerator} divided by ${denominator} plus ${globalValue}
yields ${instance.exports.divideAndAdd2(numerator, denominator)}`);
})();
```

Test this by pasting the code above in the browser console.

2.2.1.3. Implicit sections

What is important to note is that although the *global section* has disappeared in the code - even in the binary one - it's still implicitly there as whenever an *import section* imports a category C item, a C section list (**global** in this example) is implicitly created or updated :

Hence this :

```
(module
        (import                         ;; import section for a global named "addedValue" in JS-land
                "js"    "addedValue"    ;; import addedValue nested in the js field of wasmImport
                (global i32)            ;; imported as a global : constant 4-byte integer
        )                               ;; create an implicit first global section, with index 0.
        ....                            ;; other required things
)
```

is equivalent to this :

```
(module
        (global i32                     ;; implicit global section with index 0, containing the import
        ....
        )
        ....                            ;; other required things
)
```

This is the same for other import types like memory, seen in the next chapter :

```
(module
        (import                         ;; import section for a memory slab (= typed array in JS-land)
                "js"    "memory"        ;; import memory nested in the js field of wasmImport
                (memory                 ;; imported as a memory object
                ....
                )
        )                               ;; create an implicit first memory section (index 0)
)
```

...is similar to :

```
(module
        (memory  ....)                  ;; implicit memory section with index 0, containing the import
)
```

Last but not least, import sections must be defined before all the other's but the **type** one.

If the *section type* of the imported stuff exists in the module like in the following WAT code :

```
(module
        (import                         ;; import section
                "js"    "addedValue"
                (global i32)
        )
        (global i32 ....)               ;; global section
)
```

...then the real *global section* list, once completed by the imported global is :

```
(module
        (global i32 ....)       ;; imported global : index 0

        (global i32 ....)       ;; native global : index 1
)
```

Consequently, the index of the imported global is 0 and the one of the native's is 1.

2.2.2. Using HTML5 APIs and javascript functions (numerical ones)

2.2.2.1. Importing functions

For this chapter, let's consider a new - still dummy - module example where we want to use the cosine function.

```
let math = {  cos: Math.cos };
let wasmImport = { math };
let { instance } = await WebAssembly.instantiate(wasmBinary, wasmImport);
```

Notes :

- *do not forget to bound functions if necessary (not the case here) before setting them into an object (as an example* **let log = { console : console.log.bind(console) }** *if you import* **console.log***) !*
- *example chosen for simplicity's sake, but stupid because there's often an overhead whenever you cross the Wasm-host boundary (except for Firefox), so not interesting for simple functions like these that can be quickly coded using classic maths (search "Taylor series")*

Let's write a module using such functions, as an example inside a geometric function calculating the scalar product of 2 vectors from their lengths (**len1**, **len2**) and their angle (**angle**).

```
scalarProduct = len1 · len2 · cos(angle)
```

2.2.2.2. WAT code

If you're using numerical javascript function, the default type is an 8-byte float (**f64**).

```
(module
	(type                                          ;; index 0 type section : cosine function signature
		(func (param f64) (result f64))
	)
	(type                                          ;; index 1 type section : exported function signature
		(func (param f64 f64 f64) (result f64))    ;; length1, length2 and angle (radian), respectively
	)
	(import
		"math" "cos"
		(func (type 0))        ;; function with index 0 (implicit function section created)
	)
	(func (type 1)             ;; index 1 function section :
		get_local 0
		get_local 1
		f64.mul                ;; pop length values, multiply them and push result upon the stack
		get_local 2            ;; push the angle value upon the stack
		                       ;; now you have this on the stack : [..., multiplicationResult, angle]
		call 0                 ;; call the function with index 0 (imported cosine)
		                       ;; one argument -> only one pop from the stack.
		                       ;; then push the result on the stack : [..., multiplicationResult, cos(angle)]
		f64.mul                ;; pop the 2 arguments, then the stack is [ …, cosResultXMultipliedLengths]
	)
	(export "getScalarProduct" (func 1))   ;; export function with index 1, to calculate a scalar product
)
```

The logic is similar to the one of chapter 2.2.1, the only novelties are these instructions :

- **call** which… calls a function according to its index in the function section list, index given right after the instruction;
- **i32.mul** which is... the multiply operator for both sign and unsigned 4-byte integer.

2.2.2.3. Binary code

WAT token	f64	end	get_local	i32.mul
Binary token	*7C*	*0B*	*20*	*A2*

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmJsFuncExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   0D(length in bytes) 02(number of defined types : number of types sections)
   // section with index 0
   60       type 0 is a function (cosine)
   01       7C number and types of function arguments
   01       7C          number and type of returned values (!always 1 for Wasm version 1.0!)

   // section with index 1
   60       type 1 is a function (getScalarProduct)
   03       number of function arguments
      7C 7C 7C
   01       number of returned values (!always 1 for Wasm version 1.0!)
      7C    type of the returned value

02          <- Import  list : Section list for importing items from javascript
   0C(length in bytes) 01(number of import : only one, exported with the name “divide” here)
   04       import “namespace” length in byte.
   6D(m) 61(a) 74(t) 68(h)      import “namespace” : “math”
   03       import name length in bytes
   63(c) 6F(o) 73(s)            imported item: “cos”
   00       import category : define a function (category id : 0 for function, 1 for table, 2 for memory, 3 for global)
   00       type index in the type list above

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions : only one, exported with the name “divide” here)
   01       first not-imported function (index 1)  : index of the corresponding type in the type list

07          <- Export list : Section list for exporting stuffs to JS-land
   14(length in bytes) 01(number of exported items)
      10    exported item name length
      67(g) 65(e) 74(t) 53(S) 63(c) 61(a) 6C(l) 61(a) 72(r) 50(P) 72(r) 6F(o) 64(d) 75(u) 63(c) 74(t)
      00    export category : function (category id : 0 for function, 1 for table, 2 for memory, 3 for global)
      01    function index in the function list above (! index 0 is for the imported function)

0A          <- Code list : Section list for defining function bodies
   0E(length in bytes) 01(number of code sections)
      0C    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      20(get_local) 00          get the parameter at index 0 in the function signature
      20(get_local) 01          get the parameter at index 1 in the function signature
      A2                        multiply operator for, 4-byte integers
      20(get_local) 02          get the parameter at index 2 in the function signature
      10(call) 00               call function at index 0 in the function list above
      A2                        multiply operator for 4-byte integers
      0B                        notify end of code block`;
const wasmImport = { math: { cos: Math.cos } };
const { instance } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmJsFuncExample),
   wasmImport
);
const length = 1; const angle = Math.PI / 4;
const { exports: { getScalarProduct } }  = instance;
window.alert(`Wasm - ext. function : vectors of length ${length} whose angle (radian) is ${angle.toFixed(6)}
have value ${getScalarProduct(length, length, angle).toFixed(6)}`);
})();
```

Test this by pasting the code above in the browser console.

2.2.3. Generic interaction through memory

2.2.3.1. Using memory for two-way communication or string manipulation

The methods introduced in chapters 2.2.1. and 2.2.2. are not so useful if used alone :

- imported **global**s value cannot be modified from the Wasm code;
- imported functions cannot process strings because there's no default string type;
- how do we modify strings from Wasm?

Wasm memory (*memory section* in a module) comes to the rescue. *Memory sections* have the identifier with value **05**.

Under the hood, Wasm memory is basically an optimized **ArrayBuffer** whose size is a multiple of a typical chunk of memory, called a *page* and equal to 64 kBytes (65 536 bytes (2^{16})). Memory is managed linearly via *offsets* (addresses in bytes from the memory beginning).

Wasm memory is flexible as its size can grow, and can also be limited to a given page number.

*Note : if the Wasm is allowed to grow, a new **ArrayBuffer** is allocated for storing the data of the new one : the javascript reference is lost, and so all the javascript dependencies must be updated with the new one!*

Various ways can be used to create Wasm memory, by either :

1. defining a memory section in the WAT or binary code of the module (IMPORTANT : only one memory section with WebAssembly 1.0!) then exporting it for interaction;
2. creating a Wasm memory object in javascript, then importing it.

If you use method 1, the code snippets are :

- WAT :

```
(module
        (memory 1))        ;; memory section with an initial size of 1 page  (!only one section for Wasm 1.0!)
)
```

- Binary Code :

```
let wasmMemoryExampleText = `${wasmHeader}
05      <- Memory list : Section list for defining "slabs" of memory
  03    length in bytes
  01    number of defined memory slabs (always 1 for Wasm 1.0!)
  // section with index 0 (only one for Wasm 1.0!)
  00    0 if the memory can grow, 1 if the memory size is constant, 3 if memory is shared
  01    memory slab initial size in pages
`;
```

By default, if one only provides an initial size, Wasm memory is allowed to grow. For that, one has to use the **memory.grow** WAT operator (**40** in binary, this operator also pushes upon the stack the previous size) followed by the new size in pages (see chapter 2.2.3.2.1. for the javascript equivalent).

If you want to prevent that, you have to add a limit as follows :

- WAT :

```
(module
        (memory 127 1000))          ;; memory section whose size can grow from 127 pages to 1000 pages.
)
```

- Binary Code :

```
wasmMemoryExampleText = `${wasmHeader}
05          <- Memory list : Section list for defining “slabs” of memory
  05        length in bytes
  01        number of defined memory slabs (always 1 for Wasm 1.0!)
  // section with index 0
  01        0 if the memory can grow, 1 if the memory size is constant, 3 if the memory is shared
  7F        memory slab initial size in pages
  E8 07     memory slab maximum size in pages (LEB128 encoding, see Appendix A.2. for explanations)
`;
```

Finally, you can export this memory the same way you export a function :

- WAT :

```
(module
        (memory 1))                 ;; memory section with index 0
        (export "mem" (memory 0))   ;; export memory section with index 0 as “mem” variable
)
```

- Binary Code

```
const wasmMemoryExample = `${wasmHeader}
05          <- Memory list : Section list for defining “slabs” of memory
  03(length in bytes) 01(number of defined memory slabs (always 1 for Wasm 1.0!))
  // section with index 0
  00        0 if the memory can grow, 1 if the memory size is constant
  01        memory slab initial size in pages

07          <- Export list : Section list for exporting stuffs to JS-land
  07(length in bytes) 01(number of exported functions)
    03      exported item name length
    6D(m) 65(e) 6D(m)
    02      export category : memory (category id : 0 for function, 1 for table, 2 for memory, 3 for global)
    00      memory index in the memory list above (always 0 for Wasm 1.0)
`;
let { instance } = await WebAssembly.instantiate(wasmConvertDirtyHexToBinary(wasmMemoryExample));
const { exports: { mem } } = instance;
const memArray = new Uint8Array(mem.buffer);            // ArrayBuffer in mem.buffer
window.alert(`Wasm memory size is : ${memArray.length} bytes` );
```

The **ArrayBuffer** representing the Wasm memory is inside the **buffer** field of the exported memory. Test the code above by pasting the code from chapter 1.3.5 into the browser console, then the one in the binary code section right above.
In the next, we will introduce how to process strings in Wasm. The example will be given for imported memory (method 2), but this could have been done with method 1.

2.2.3.2. Accessing the DOM from a Wasm module

This is quite a relatively complicated chapter - do not hesitate to play with the code to become familiar with it - whose main goal is to allow you to understand the process. Indeed, DOM manipulation is still way easier in Javascript, so only here to let your understand the principle!

2.2.3.2.1. Setting up the memory

It's convenient to setup the memory at runtime, with a logic similar to the one for importing globals (chapter 2.2.1.1.), as follows :

```
// 64 Megabytes (one page = 64kB)  allocated to store lots of text!
let memoryDescriptor = { initial: 1024, maximum: undefined };
let memory = new WebAssembly.Memory(memoryDescriptor);      // Wasm memory object
```

In this case, where no maximum is set in **memoryDescriptor**, memory can be grown from javascript like this :

```
memory.grow(512);                    // increasing the memory by 512 pages (new size will be 96 Megabytes)
```

2.2.3.2.2. Building a Wasm interface for using strings

Say we want to create HTML tags in Wasm : we need to create first a tiny API called **domApi** operating purely on string arguments instead of objects :

```
const domApi = (document => {                    // IIFE (see chapter 1.3.2. )
   const createElement = document.createElement.bind(document);
   const querySelector = document.querySelector.bind(document);
   let $elem;

   const create = function createAndAppend($parent='body', content='', tag='div', id='', classes='') {
      $parent = querySelector($parent);
      $elem = createElement(tag);
      $elem.textContent = content;
      $elem.id = id;
      $elem.class = classes;
      $parent.appendChild($elem);
   };

   const update = function findAndChange(selector, content='') {
      $elem = querySelector(selector);
      if ($elem)    $elem.textContent = content;
   };

   const remove = function findAndDelete(selector, content='') {
      $elem = querySelector(selector);
      if ($elem)    $elem.parentNode.removeChild($elem);
   };

   const setAttribute = function setAttribute(selector, attributeName='', attributeValue='') {
      $elem = querySelector(selector);
      $elem.setAttribute(attributeName, attributeValue);
   };

   const getAttribute = function getAttribute(selector, attributeName='') {
      $elem = querySelector(selector);
      return $elem.getAttribute(attributeName);
   };

   return { create, update, remove, setAttribute, getAttribute };
})(document);
```

Then, we must create another small API called **stringApi** mapping strings to values corresponding to string start locations (addresses) inside our Wasm memory *aka* **offset**s. Beware, by personal decision the first letter of a string starts 4 bytes after its "starting location". Indeed, this space is used to store an integer specifying string length in bytes.

```
const memory = { buffer: new Uint8Array(100 * 2**16).buffer };
const stringApi = (memory => {          // IIFE : memory is our Wasm memory object (with a buffer field)
   const maxSize = new Uint8Array(memory.buffer).length;
   const encoder = new TextEncoder();              // UTF-8 by default
   const enc = encoder.encodeInto.bind(encoder);
   const decoder = new TextDecoder();              // UTF-8 by default
   const dec = decoder.decode.bind(decoder);
   let size, previousOffset, currentOffset = 0, nextOffset;  // respectively string size; last, current and future string addresses
   const map = {};  // for testing purpose : to store the correspondence between strings and offsets

   const encode = function utf8Encode(string, offset=currentOffset) {
       if (map[string]) {                          // optimization
         return map[string];
       }
       size = new Blob([string]).size;     // an UTF-8 character can take several bytes so do not use string.length!
      // next offset is N bytes furthers, with N multiple of 4 (otherwise Uint32Array cannot be used 4 lines below)...
      // … and with length equals size S of the string plus 4 bytes to store the size (see 4 lines below)
       nextOffset = currentOffset + 4 * (1 + Math.ceil(size / 4));
       if (offset % 4)    throw new Error('offset must be a multiple of 4!');
       if (size > maxSize || nextOffset > maxSize)    throw new Error('String is too big!');
       new Uint32Array(memory.buffer, offset, 4)[0] = size;   // 4 first bytes reserved for storing the string size (in bytes)
       enc(string, new Uint8Array(memory.buffer, 4 + offset) );
       map[string] = offset;
       previousOffset = offset;
       currentOffset = nextOffset;
       return offset;
   };

   const decode = function utf8Decode(offset=previousOffset) {
      [size] = new Uint32Array(memory.buffer, offset, 1);
      return dec( new Uint8Array(memory.buffer, offset + 4, size) );
   };
   // used after growing the memory,  for keeping the memory reference (previous reference lost)
   const setMemory = (newMemory) => { memory = newMemory };

   return { encode, decode, setMemory, map };
})(memory);
```

Finally, one must use **stringApi** so that **domApi** becomes Wasm compatible by using number (offsets) instead of strings :

```
function mapStringsToOffsets(func) {              // func is as an example domApi.create
   // encode the strings inside the memory
   return (...offsetArguments) => {       // returned function is for instance domApi.create, accepting offsets instead of strings
      stringArguments = offsetArguments.map( offset => stringApi.decode(offset) );
      const result = func.apply(this, stringArguments);
      return (typeof result === 'string')              // convert the result to an offset if this is a string
          : stringApi.encode(result)
          ? result
   }
}
for (let key in domApi) {       // we overwrite the domApi so that its functions accept offsets as arguments.
   domApi[key] = mapStringsToOffsets(domApi[key]);
}
```

2.2.3.2.3. Code summary

```
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;

let memoryDescriptor = { initial: 100 };
let memory = new WebAssembly.Memory(memoryDescriptor);
const domApi = (() => {
   let $elem;
   const querySelector = document.querySelector.bind(document);
   const create = function createAndAppend($parent='body', content='', tag='div', id='', classes='') {
      $parent = querySelector($parent);
      $elem = document.createElement(tag);
      $elem.textContent = content;      $elem.id = id;      $elem.class = classes;
      $parent.appendChild($elem);
   };
   const update = function findAndChange(selector, content='') {
      $elem = querySelector(selector);
      if ($elem)    $elem.textContent = content;
   };
   const remove = function findAndDelete(selector, content='') {
      $elem = querySelector(selector);
      if ($elem)       $elem.parentNode.removeChild($elem);
   };
   const setAttribute = (selector, name='', value='') => querySelector(selector).setAttribute(name, value);
   const getAttribute = (selector, name='') => querySelector(selector).getAttribute(name);
   return { create, update, remove, setAttribute, getAttribute };
})();

const stringApi = ((memory) => {
   const maxSize = new Uint8Array(memory.buffer).length;
   const encoder = new TextEncoder();             const enc = encoder.encodeInto.bind(encoder);
   const decoder = new TextDecoder();             const dec = decoder.decode.bind(decoder);
   let size, previousOffset, currentOffset = 0, nextOffset, map = {};
   const encode = function utf8Encode(string, offset=currentOffset) {
       if (map[string])    return map[string];
       size = new Blob([string]).size;
       nextOffset = currentOffset + 4 * (1 + Math.ceil(size / 4));
       if (offset % 4)    throw new Error('offset must be a multiple of 4!');
       if (size > maxSize || nextOffset > maxSize)    throw new Error('Memory is depleted!');
       new Uint32Array(memory.buffer, offset, 4)[0] = size;
       enc(string, new Uint8Array(memory.buffer, 4 + offset) );
       map[string] = offset;
       previousOffset = offset;
       currentOffset = nextOffset;
       return offset;
   };
   const decode = function utf8Decode(offset=previousOffset) {
      [size] = new Uint32Array(memory.buffer, offset, 1);
      return dec( new Uint8Array(memory.buffer, offset + 4, size) );
   };
   const setMemory = (newMemory) => { memory = newMemory };
   return { encode, decode, setMemory, map };
})(memory);

const mapStringsToOffsets = func => (...offsetArguments) => {
   stringArguments = offsetArguments.map( offset => stringApi.decode(offset) );
   return stringApi.encode( func.apply(this, stringArguments) );
}
for (let key in domApi)         domApi[key] = mapStringsToOffsets(domApi[key]);
```

2.2.3.3. Using strings in Wasm

With the code from the last chapter, we're ready to use strings inside Wasm, provided the exported functions do the opposite process by decoding the offsets into strings by using the function below (simply the reverse of **mapStringsToOffsets**) :

```
const mapOffsetsToStrings = func => (...stringArguments) => {
   offsetArguments = stringArguments.map( string => stringApi.encode(string) );
   return stringApi.decode( func.apply(this, offsetArguments) );
};
```

One imports both **memory**, the Wasm memory object (to interact with the string in Wasm in the next chapter) and **domApi** :

```
const wasmImport = { domApi, string: { memory } };
let { instance } = await WebAssembly.instantiate(wasmBinary, wasmImport);
```

Finally one transforms the exported function (**modifyDOM**) to accept strings instead of offsets :

```
const modifyDOM = mapOffsetsToStrings(instance.exports.modifyDOM);
```

For simplicity's sake, the Wasm **modifyDOM** function chosen as an example has the same signature as **stringApi.create** :

- it creates a new div with an identifier of value 'testId';
- it changes the style attribute so that the text appears in red.

2.2.3.3.1. WAT code

```
(module
        ;; type with index 0 in type list, as a signature for both stringApi.create and modifyDOM
        (type ( func (param i32 i32 i32 i32 i32) ) )
        ;; type with index 1 in type list, as a signature for stringApi.setAttribute
        (type ( func (param i32 i32 i32) ) )
        (import "string" "memory" (memory 100) )
        (import "domApi" "create" ( func (type 0) ) )              ;; function with index 0 in function list
        (import "domApi" "setAttribute" ( func (type 1) ) )        ;; function with index 1 in function list
        (func (type 0)                                             ;; function with index 2 in function list
                get_local 0
                get_local 1
                get_local 2
                get_local 3
                get_local 4
                call 0              ;; create new DOM node (call "create")
                i32.const 0         ;; offset for '#testId'
                i32.const 12        ;; offset for 'style'
                i32.const 24        ;; offset for 'color:red'
                call 1              ;; change style attribute (call "setAttribute")
        )
        (export "modifyDOM" (func 2))
)
```

2.2.3.3.2. Binary code

Test this by pasting the code from chapter 2.2.3.2.3. then the one below in the console :

```
(async () => {
stringApi.encode('#testId');                    // offset: 0
stringApi.encode('style');                      // offset: 12
stringApi.encode('color:red                ');  // offset: 24
const wasmStringExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
  0F(length in bytes) 02(number of defined types : number of types sections)
  60    define a function type for type with index 0
  05    7F 7F 7F 7F 7F      number of function arguments and types
  00    number of returned values

  60    define a function type for type with index 1
  03    7F 7F 7F            number of function arguments and types
  00    number of returned values

02          <- Import  list : Section list for importing items from javascript
  38(length in bytes) 03(number of imported items)
  06(string length) 73(s) 74(t) 72(r) 69(i) 6E(n) 67(g)          import “namespace” : “string”
  06(string length) 6D(m) 65(e) 6D(m) 6F(o) 72(r) 79(y)          imported item : “memory”
  02    import category : memory
  00    0 = memory can grow, 1 = memory limited (maximum)
  64    initial memory size in pages
  …     if there’s a maximum, its value goes there.

  06(string length) 64(d) 6F(o) 6D(m) 41(A) 70(p) 69(i)          “domApi”
  06(string length) 63(c) 72(r) 65(e) 61(a) 74(t) 65(e)          “create”
  00    import category : function
  00    type index in the type list above

  06(string length) 64(d) 6F(o) 6D(m) 41(A) 70(p) 69(i)                                  “domApi”
  0C(string length) 73(s) 65(e) 74(t) 41(A) 74(t) 74(t) 72(r) 69(i) 62(b) 75(u) 74(t) 65(e)   “setAttribute”
  00    import category : function
  01    type index in the type list above

03          <- Function list : Section list for defining functions
  02(length in bytes) 01(number of defined functions)
  00    first not-imported function (index 2): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
  0D(length in bytes) 01(number of exported functions)
    09(string length) 6D(m) 6F(o) 64(d) 69(i) 66(f) 79(y) 44(D) 4F(O) 4D(M)          “modifyDOM”
    00  export category : function
    02  function index in the function list above (indices 0 and 1 are for the imported ones!)

0A          <- Code list : Section list for defining function bodies
  18(length in bytes) 01(number of code sections)
    16  length in byte of the code section below
    00  number of local variables other than arguments (none here)
    20(get_local) 00 20(get_local) 01 20(get_local) 02 20(get_local) 03 20(get_local) 04   prepare the stack
    10(call) 00             execute function with index 0 (domApi.create) with the 5 values upon the stack
    41 00                   push 4-float integer with value 0 upon the stack (offset for ‘#testId’)
    41 0C                   push 4-float integer with value twelve upon the stack (offset for ‘style’)
    41 18                   push 4-float integer with value twenty-four upon the stack (offset for ‘color:red      ’)
    10(call) 01             execute function with index 1 (domApi.setAttribute) with the 3 values upon the stack
    0B                      notify end of code block`;
const wasmImport = { domApi, string: { memory } };
const { instance } = await WebAssembly.instantiate(
  wasmConvertDirtyHexToBinary(wasmStringExample),
  wasmImport
);
const mapOffsetsToStrings = func => (...stringArguments) => {
  offsetArguments = stringArguments.map( string => stringApi.encode(string) );
  return stringApi.decode( func.apply(this, offsetArguments) );
};
modifyDOM = mapOffsetsToStrings(instance.exports.modifyDOM);
modifyDOM('body', 'Accessing the DOM from a Wasm module!', 'div', 'testId', '');
})();
```

2.2.3.4. Interlude - Limitations

As you notice, it would be a lie to tell that one can conveniently access the DOM in Wasm, because our dummy example clearly shows some limitations :

1. memory manually managed;
2. simple calls of DOM functions inside Wasm are not efficient because the functions must be wrapped then unwrapped (see **mapStringsToOffsets** and **mapOffsetsToStrings**) to accept as arguments in one case memory offsets and in the other's strings. Hence each call executes a javascript wrapper so the irony is that native javascript is faster if Wasm do not perform intensive work during the call to justify this extra cost.
3. Why importing the memory if one doesn't modify strings stored in it in Wasm?

First and second points are out of the scope of this book, and are some of the reasons why one should use Emscripten as an abstraction for big projects.

Concerning point 3, you can indeed omit to import the memory and observe the same result if you don't want to interact with it in Wasm, by replacing in the code of the last chapter the few lines below :

```
02          <- Import  list : Section list for importing items from javascript
  38(length in bytes) 03(number of imported items)
  06(string length) 73(s) 74(t) 72(r) 69(i) 6E(n) 67(g)          import "namespace" : "string"
  06(string length) 6D(m) 65(e) 6D(m) 6F(o) 72(r) 79(y)          imported item : "memory"
  02        import category : memory
  00        0 = memory can grow, 1 = memory limited (maximum)
  64        initial memory size in pages
  …         if there's a maximum, its value goes there.
```

by

```
02          <- Import  list : Section list for importing items from javascript
  27(length in bytes) 02(number of imported items)
```

2.2.3.5. Modify memory directly in Wasm

Let's now modify directly a string - stored in our memory - with Wasm instructions. As an example, one will change the color of the new DOM element to blue instead of red. Hence, one replaces 4 latin characters (1 byte per character) in the string at offset 24, that is to say starting at the 24th byte from the beginning of the memory (look back to the first lines of chapter 2.2.3.3.2.) :

'color:red ' becomes 'color:blue '

Why replacing only 4 bytes? Because memory instructions can process 4 bytes or 8 bytes at once so 4 bytes chosen for simplicity's sake. Offset 24 in the Wasm memory corresponds to the length (stored in 4 bytes) of 'color:red '. So the color value string actually starts 4 bytes further, at offset 28 (letter 'c' of color:red), and the color to replace starts 6 bytes even further, at offset 34 (**22** in hexadecimal).

Then, these 4 letters must be encoded as a single 4-byte value that can be processed by Wasm **i32** operators. This will be by no means a small value, so LEB128 compression is mandatory (look back at **encodeLeb** at chapter 2.3.1. if necessary) :

```
const integerWithBlue = '62 6C 75 65';          // 4 bytes corresponding to the letters b l u e

function convert4Utf8BytesToWasm(utf8WordWithSpace) {
    const hexString = utf8WordWithSpace.split(' ').reverse().join('');          // reverse the order (little endian)
    const word =  parseInt(hexString, 16);
    console.log(word);                          // 1702194274 in our case
    return encodeLeb(word);
}
const blueLeb = convert4Utf8BytesToWasm(integerWithBlue);      // 5 LEB-bytes : "E2 D8 D5 AB 06"
```

2.2.3.5.1. WAT code

The code is just a tweak of the previous one, by overwriting bytes in the **'color:red '** string before calling the **setAttribute** method for the style of the new DOM element. That for, one uses the **i32.store** operator, processing 2 stack values : the first one is the offset from the beginning of the memory where an **i32** value must be replaced, the second one is the new i32 value. In addition, the store instruction accepts 2 additional parameters, right after the **i32.store** operator (0 0 in the example), explained in chapter 2.2.3.6.3.

```
(module
        (type ( func (param i32 i32 i32 i32 i32) ) )
        (type ( func (param i32 i32 i32) ) )
        (import "string" "memory" (memory 100) )
        (import "domApi" "create" ( func (type 0) ) )              ;; function with index 0 in function list
        (import "domApi" "setAttribute" ( func (type 1) ) )        ;; function with index 1 in function list
        (func (type 0)                                             ;; function with index 2 in function list
                get_local 0
                get_local 1
                get_local 2
                get_local 3
                get_local 4
                call 0              ;; create new DOM node
                i32.const 34        ;; offset corresponding to the second 'r' letter  'color:red '
                i32.const 1702194274        ;; 32-bit value to be replaced in the memory at the offset 34
                i32.store 0 0               ;; store value at offset 34 (0 0 are alignment and relative offset
                                    ;; see chapter 2.2.3.6.3. for explanations about these values)
                i32.const 0         ;; offset for '#testId'
                i32.const 12        ;; offset for 'style'
                i32.const 24        ;; offset for 'color:red    ', changed to 'color:blue    '
                call 1              ;; change style attribute
        )
        (export "modifyDOM" (func 2))
)
```

Note : the astute reader should have noted that we don't pass the index of the memory operator to the **i32.store** *memory operator. This is not a problem for Wasm 1.0 because there must at most one memory section so implicit reference, but one should expected some change for these operator (and the binary code in the next chapter) for future versions of WebAssembly accepting multiple memories per module.*

2.2.3.5.2. Binary code

Also, the binary code is just a variation of the previous code, concerning only the code list :

WAT token	i32.const	i32.store	i32.load
Binary token	*41*	*36*	*28*

```
0A          <- Code list : Section list for defining function bodies
   23(length in bytes) 01(number of code sections)
      21    length in byte of the code section below
      00    number of local variables (none here)
      20(get_local) 00 20(get_local) 01 20(get_local) 02 20(get_local) 03 20(get_local) 04     prepare the stack
      10(call) 00                 execute function with index 0 (domApi.create) with the 5 values upon the stack
      41 22                       offset thirty-four corresponds to the start of 'red' in 'color:red                '
      41 ${blueLeb}               replace substring 'red ' by 'blue', see JS code above for blueLeb derivation
      36 00 00                    store 4-byte value at byte thirty-four, with null alignment and relative offset,
respectively
      41 00                       push 4-float integer with value 0 upon the stack (offset for '#testId')
      41 0C                       push 4-float integer with value twelve upon the stack (offset for 'style')
      41 18                       push 4-float integer with value twenty-four upon the stack (offset for 'color:red      ')
      10(call) 01                 execute function with index 1 (domApi.setAttribute) with the 3 values upon the stack
      0B                          notify end of code block
```

2.2.3.5.3. Memory access tweaks

Memory is accessed via *load* (read) and *store* (write) operators, which both accepts 2 optional (equal to zero if not provided), non-stack arguments which are respectively the memory *alignment* and the *relative offset* (ex: **i32.load** *alignment relativeOffset*):

- The relative offset is an extra value to be added to the offset (first stack argument of load or store) to obtain the real offset. It's not used in this book, but its usefulness can be explained as an example if a **i32** sequence contains composite data, such as a player list where each player is modeled by an **i32**, and where 1st byte holds its experience, the 2nd one its life, the 3rd its ammunitions. Say **i32.load8_u** is the operator, similar to **i32.load**, but to read a byte instead of 4 bytes at once, then to read the life of player A you don't need :

```
i32.const offsetToByteWithLifeOfPlayerA
i32.load8_u       0         0         ;; life of player A put upon the stack, casted to an i32
                  ;;align   ;;relative offset
```

...but you can simply use the offset corresponding to player A, and keep the byte corresponding to the life an implementation detail :

```
i32.const offsetToPlayerA             ;; first byte is experience, second one is life, third ammunitions
i32.load8_u       0         1         ;; life of player A put upon the stack, casted to an i32)
                  ;;align   ;;offset to add to read life
```

Similarly, to read the ammunitions :

```
i32.const offsetToPlayerA
i32.load8_u       0         2
                  ;;align   ;;offset to add to read ammunitions
```

- Alignment is more tricky : do not use a value different from zero if you don't fully understand what you're doing, otherwise there's a significant performance penalty (especially if memory is accessed inside huge loops).

  Processors (CPUs) treat bytes per groups, by numbers that are powers of 2 :
  2^0= 1 byte, 2^1= 2 byte, 2^2= 4 bytes (**i32**/**f32**), 2^3= 8 bytes (**i64**/**f64**), 2^4 = 16 bytes (**v128**)

  When a CPU processes a value composed of several bytes, it works way faster when the bytes are aligned, that is to say when the starting and ending offsets of the considered value are multiples of its size. Let's use a metaphor : think about a commis de cuisine who's used to take 4 dishes (our **i32**, 4-byte value, a dish "equals" a byte) at once provided they're all on the same table among a row of separated tables (our memory). Also imagine that the person working before the commis and setting the dishes on the row of separated tables (the memory setter) hasn't aligned the 4 related dishes (2 on a table, 2 on another's). Consequently, the commis de cuisine will work less efficiently (moving around for less than 4 dishes instead of taking the 4 ones at once).

  So concretely the alignment value (the exponent E) is only a hint given to the Wasm virtual machine, which means that one guarantees at runtime this memory alignment (starting and ending offsets of a value are multiples 2^E bytes) for the provided offsets and values :
  - If this is the case, the Wasm virtual machine can perform various machine-code optimizations, yielding extra performance (the higher the alignment, the better);
  - If this is not the case, the Wasm virtual machine will lose a lot of time due to unoptimized memory access, even more time than if alignment value is zero (default alignment).

2.2.3.5.4. Code summary

Test the string modification in Wasm by pasting the 2 next pages into your console :

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
let memory = new WebAssembly.Memory({ initial: 100 });
const domApi = (() => {
   let $elem;          const querySelector = document.querySelector.bind(document);
   const create = ($parent='body', content='', tag='div', id='', classes='') => {
      $parent = querySelector($parent);
      $elem = document.createElement(tag);
      $elem.textContent = content;      $elem.id = id;      $elem.class = classes;
      $parent.appendChild($elem);
   };
   const setAttribute = (selector, name='', value='') => querySelector(selector).setAttribute(name, value);
   return { create, setAttribute };
})();
const stringApi = (memory => {
   const maxSize = new Uint8Array(memory.buffer).length;
   const encoder = new TextEncoder();              const enc = encoder.encodeInto.bind(encoder);
   const decoder = new TextDecoder();              const dec = decoder.decode.bind(decoder);
   let size, previousOffset, currentOffset = 0, nextOffset, map = {};
   const encode = (string, offset=currentOffset) => {
      if (map[string])    return map[string];
      size = new Blob([string]).size;
      nextOffset = currentOffset + 4 * (1 + Math.ceil(size / 4));
      if (offset % 4)    throw new Error('offset must be a multiple of 4!');
      if (size > maxSize || nextOffset > maxSize)    throw new Error('Memory is depleted!');
      new Uint32Array(memory.buffer, offset, 4)[0] = size;
      enc(string, new Uint8Array(memory.buffer, 4 + offset) );
      map[string] = offset;    previousOffset = offset;      currentOffset = nextOffset;
      return offset;
   };
   const decode = (offset=previousOffset) => {
      [size] = new Uint32Array(memory.buffer, offset, 1);
      return dec( new Uint8Array(memory.buffer, offset + 4, size) );
   };
   return { encode, decode };
})(memory);
const mapStringsToOffsets = func => (...offsetArguments) => {
   stringArguments = offsetArguments.map( offset => stringApi.decode(offset) );
   return stringApi.encode( func.apply(this, stringArguments) );
}
for (let key in domApi)         domApi[key] = mapStringsToOffsets(domApi[key]);
function encodeLeb(value) {              // see chapter 2.2.3.1
   let more = true;  let i = 0;  const signBit = 64; const mask7Bits = 127;
   let bytes = [null, null, null, null, null, null, null, null, null, null];
   while (more) {
      bytes[i] = value & mask7Bits;
      value >>= 7;
      if (  ( !value && !(bytes[i] & signBit) ) || ( value == -1 && (bytes[i] & signBit) )  ) {
         more = false;
      } else {
         bytes[i] |= 128;
      }
      bytes[i] = bytes[i].toString(16).padStart(2, '0').toUpperCase();
      i++;
   }
   return bytes.filter(group => group !== null).join(' ');
};
```

```
const integerWithBlue = '62 6C 75 65';              // 4 bytes corresponding to the letters b l u e
function convert4Utf8BytesToWasm(utf8WordWithSpace) {
    const hexString = utf8WordWithSpace.split(' ').reverse().join('');              // reverse the order (little endian)
    return encodeLeb( parseInt(hexString, 16) );
}
const blueLeb = convert4Utf8BytesToWasm(integerWithBlue);      // 5 LEB-bytes
stringApi.encode('#testId');  stringApi.encode('style');     stringApi.encode('color:red                  ');
const wasmStringExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
  0F(length in bytes) 02(number of defined types : number of types sections)
  60      define a function type for type with index 0
  05      7F 7F 7F 7F 7F      number of function arguments and types
  00      number of returned values
  60      define a function type for type with index 1
  03      7F 7F 7F            number of function arguments and types
  00      number of returned values

02          <- Import  list : Section list for importing items from javascript
  38(length in bytes) 03(number of imported items)
  06(string length) 73(s) 74(t) 72(r) 69(i) 6E(n) 67(g)          import “namespace” : “string”
  06(string length) 6D(m) 65(e) 6D(m) 6F(o) 72(r) 79(y)          imported item : “memory”
  02      import category : memory
  00      0 = memory can grow, 1 = memory limited (maximum)
  64      initial memory size in pages

  06(string length) 64(d) 6F(o) 6D(m) 41(A) 70(p) 69(i)          import “namespace” : “domApi”
  06(string length) 63(c) 72(r) 65(e) 61(a) 74(t) 65(e)          imported item : “create”
  00      import category : function
  00      type index in the type list above

  06(string length) 64(d) 6F(o) 6D(m) 41(A) 70(p) 69(i)                                  “domApi”
  0C(string length) 73(s) 65(e) 74(t) 41(A) 74(t) 74(t) 72(r) 69(i) 62(b) 75(u) 74(t) 65(e)      “setAttribute”
  00      import category : function
  01      type index in the type list above

03          <- Function list : Section list for defining functions
  02(length in bytes) 01(number of defined functions)
  00      first not-imported function (index 2): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
  0D(length in bytes) 01(number of exported items)
    09(string length) 6D(m) 6F(o) 64(d) 69(i) 66(f) 79(y) 44(D) 4F(O) 4D(M)                  “modifyDOM”
    00    export category : function
    02    function index in the function list above (indices 0 and 1 are for the imported ones!)

0A          <- Code list : Section list for defining function bodies
  23(length in bytes) 01(number of code sections)
    21    length in byte of the code section below
    00    number of local variables other than arguments (none here)
    20(get_local) 00 20(get_local) 01 20(get_local) 02 20(get_local) 03 20(get_local) 04    prepare the stack
    10(call) 00                 execute function with index 0 (domApi.create) with the 5 values upon the stack
    41 22                       offset thirty-four corresponds to the start of ‘red’ in 'color:red                  '
    41 ${blueLeb}               replace substring ‘red ’ by ‘blue’, see JS code above for blueLeb derivation
    36 00 00                    store the 4-byte value at thirty-four byte from the memory beginning
    41 00                       push 4-float integer with value 0 upon the stack (offset for ‘#testId’)
    41 0C                       push 4-float integer with value twelve upon the stack (offset for ‘style’)
    41 18                       push 4-float integer with value twenty-four upon the stack (offset for ‘color:red      ’)
    10(call) 01                 execute function with index 1 (domApi.setAttribute) with the 3 values upon the stack
    0B                          notify end of code block`;
let { instance: { exports: { modifyDOM } } } = await WebAssembly.instantiate(
  wasmConvertDirtyHexToBinary(wasmStringExample),
  { domApi, string: { memory } }
);
const mapOffsetsToStrings = func => (...stringArguments) => {
  offsetArguments = stringArguments.map( string => stringApi.encode(string) );
  return stringApi.decode( func.apply(this, offsetArguments) );
};
modifyDOM = mapOffsetsToStrings(modifyDOM);
modifyDOM('body', 'Color set in blue in the Wasm module!', 'div', 'testId', '');
})();
```

2.2.3.6. Direct memory initialization

If you need to initially populate Wasm memory with values different from 0, you're not required to do this from Javascript (initialize memory object arraybuffer then import it). Instead, you can directly perform it from Wasm through a *data* section. A *data section* has the identifier with value 11.

When you initialize memory in a data section, you must give a byte sequence, and also an offset (**i32.const** operator followed by an integer) at which the memory bytes must be replaced by the byte sequence. Hence, bytes before the offset are not affected!

2.2.3.6.1. WAT Code

Default values are UTF-8 characters, so if you want to define numerical values it's more explicit to use hexadecimal byte notation prepended with backslash (ex: "\ff" for unsigned byte value 255). Also if you use numerical values, do not forget the little-endian architecture ! Hence, value 3 stored as an i32 (4 bytes) is **"\03\00\00\00"** (low byte first) instead of ~~**"\00\00\00\03"**~~.

For multiline memory declaration or clarity, you can split your string into multiple ones (syntactic sugar as there's still a single byte sequence) :

"\03\00\00\00div" is equivalent to ***"\03\00\00\00" "div"***

Say that we want to define the "div" string inside our module by following the same principle as in the previous chapter (string prepend by its size in byte in the memory, as an **i32**), then one can write this simple test module :

```
(module
	(type                                   ;; type  with index 0 in type list
		(func (param i32) (result i32))     ;; argument is offset, result is …
	)                                       ;; … unsigned byte value at this offset (casted to i32)
	(memory 1 1)                ;; 64 kB (1 page) memory, also limited to 1 page (cannot be grown).
	(data
		(i32.const 0)           ;; starting offset for memory initialization
		"\03\00\00\00"          ;; first 4 bytes will correspond to the byte length (i32) of the "div" string
		"div"                   ;; following bytes are filled by these 3 UTF-8 characters
	)
	(func (type 0)              ;; function with index 0 in function list and signature with index 0 in type list
		get_local 0             ;; push the argument (offset) upon the stack
		i32.load8_u             ;; load memory value at offset 0 as unsigned byte casted to an i32
	)
	(export "testData" (func 0))
)
```

2.2.3.6.2. Binary code / code summary

WAT token	i32.const	i32.store	i32.load	i32.load8_u
Binary token	*41*	*36*	*28*	*2D*

Test this by pasting the code below in the browser console.

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;

wasmDataExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   06(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   01       7F          number of function arguments and types
   01       7F          number and type of returned values

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first not-imported function (index 2): index of the corresponding type in the type list above

05          <- Memory list : Section list for defining “slabs” of memory
   04(length in bytes) 01(number of defined memory slabs (always 1 for Wasm 1.0!))
   // section with index 0
   01       0 if the memory can grow, 1 if the memory size is constant, 3 if the memory is shared
   01       memory slab initial size in pages
   01       memory slab maximum size in pages

07          <- Export list : Section list for exporting stuffs to JS-land
   0C(length in bytes) 01(number of exported functions)
      08(string length in bytes) 74(t) 65(e) 73(s) 74(t) 44(D) 61(a) 74(t) 61(a)                “testData”
      00    export category : define a function export (category id : 0 for function, 1 for table, 2 for memory, 3 for
global)
      00    function index  in the function list above

0A          <- Code list : Section list for defining function bodies
   09(length in bytes) 01(number of code sections)
      07    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      20(get_local) 00                    offset of the “div” string length'
      2D(load byte)  00 00        load byte whose offset is stack top value, with zero alignment and relative offset
      0B                          notify end of code block

0B          <- Data list : Section list for initializing a memory section
   0D(length in bytes) 01(number of data sections (always 1 for Wasm 1.0!))
   00       index of the memory section to be initialized  (always 1 for Wasm 1.0!)
   // section with index 0 (only this)
   41(const) 00          offset at which the initialization start (bytes before are not touched)
   0B(end)               as offset are declared with an instruction block (const operator above), the end must be specified
   07(byte sequence length) 03 00 00 00 64(d) 69(i) 76(v)        byte sequence starting at offset above
`;
let { instance: { exports: { testData } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmDataExample),
);
window.alert(`Wasm - Initialization : Memory bytes at offsets 0 and 4 (first letter, “d”) have unsigned values
${testData(0)} and ${testData(4)} (should be 100 for “d” (64 in hexadecimal)), respectively.`);
})();
```

2.2.4. Automatically starting a function via a start section

As said in the introduction, there's no main function inside a Wasm module. However, you can choose the function to be automatically executed - as soon as a module is instantiated - through a single *start section*. A *start section* has the identifier with value 08.

This example is stupid because we simply run an imported function, but you can imagine far more useful examples once you've finished this ebook.

```
const test = () => window.alert(`Wasm : this function is automatically started by the Wasm module!`);
let wasmImport = { dom: { alert:  test } };
let { instance } = await WebAssembly.instantiate(wasmBinary, wasmImport);
```

2.2.4.1. WAT code

For once the code is straightforward :

```
(module
        (type                                   ;; index 0 type section : window.alert with no arguments
                (func)
        )
        (import
                "dom" "alert"
                (func (type 0))         ;; function with index 0 (implicit function section created)
        )
        (start 0                        ;; one simply gives the function index for automatically start
        )
)
```

2.2.4.2. Binary code / code summary

Test this by pasting the code below in the browser console.

```
(async () =>  {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmStartModule = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   04(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   00       (no arguments)
   00       (no returned value (DOM function))
02          <- Import  list : Section list for importing items from javascript
   0D(length in bytes) 01(number of defined types : number of types sections)
   Imported function has index 0
   03(length in bytes) 64(d) 6F(o) 6D(m)                          import “namespace” : “dom”
   05(length in bytes) 61(a) 6C(l) 65(e) 72(r) 74(t)              imported item : “alert”
   00       import category : define a function (category id : 0 for function, 1 for table, 2 for memory, 3 for global)
   00       type index in the type list above
08          <- Start  list : Section list for automatically starting functions
   01(length in bytes)
   00                  index of the function to be run as soon as the module is instantiated!
`;

const test = () => window.alert(`Wasm - Autostart : This dialog is triggered from the Wasm module!`);
const  wasmImport = { dom: { alert:  test } };
await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmStartModule),
   wasmImport
);
})();
```

3. Structured control flows

This chapter will teach you how to write real-world functions by controlling the execution flow.

3.1. Conditions

Control flow is often built upon variables used as conditions. In Wasm, such variables must be of **i32**-type. If your condition depends on a comparison, you can use binary operators, pushing 1 upon the stack if the condition is true, 0 otherwise :

	===	!==	Signed <	Unsigned <	Signed >	Unsigned >	Signed <=	Unsigned <=	Signed >=	Unsigned >=
WAT	i32.eq	i32.ne	i32.lt_s	i32.lt_u	i32.gt_s	i32.gt_u	i32.le_s	i32.le_u	i32.ge_s	i32.ge_u
Binary	*46*	*47*	*48*	*49*	*4A*	*4B*	*4C*	*4D*	*4E*	*4F*

3.2. If/Else

The if/else pattern is classic (pseudo WAT code) :

```
;; last value upon the stack will be used as the condition (i32)
if                  ;; test the last stack value
        ….          ;; instructions executed if i32 value is different from zero
else
        ….
end     ;; end of the If/Else code block
```

Let's write a function called "testIf" returning 1 if the argument is not empty, 0 otherwise.

3.2.1. WAT code

One can cast a value into a valid Wasm condition using the **i32.trunc_s/X** operator (**X** replaced by the original type, e.g. **f64**). The **i32** value upon the stack is then popped by the **if** operator. The **else** operator is not mandatory. If a value is pushed upon the stack while in an if/else block, one must specify a **result** followed by its type between the parenthesis right after the **if** operator (e.g. **(result i32)**).

```
(module
        (type                                   ;; type  with index 0 in type list
                (func (param f64) (result i32))         ;; f64 argument for default JS numeric value
        )
        (func (type 0)                  ;; function with index 0 in function list and signature with index 0 in type list
                get_local 0
                i32.trunc_s/f64         ;; unary operator to cast f64 float (from get_local) to signed i32
                if (result i32)         ;; if/else execution results in an i32 value pushed upon the stack…
                                        ;; .... if last value on the stack is different from 0 (only i32 are accepted!)
                        i32.const 1
                else
                        i32.const 0
                end
        )
        (export "testIf" (func 0))
)
```

If you want to invert this condition (like the ! operator in javascript), you can use the **i32.eqz** operator right after the condition (after **i32.trunc_s/f64** here) that consumes the last value upon the stack and replaces it by 1 if not null and 0 otherwise.

3.2.2. Binary code

WAT token	i32.const	i64.trunc_s/f64	f64 type	i32.eqz	if	(if IF pushes nothing upon the stack or void)	else	end
Binary token	*41*	*AA*	*7C*	*45*	*04*	*40*	*05*	*0B*

01 **<- Type list : Section list for defining types (parameters and results)**
06(length in bytes) **01**(number of defined types : number of types sections)
60 define a function type for type with index 0
01 ***7C*** number of function arguments and types
01 ***7F*** number of returned value and type

03 **<- Function list : Section list for defining functions**
02(length in bytes) **01**(number of defined functions)
00 first not-imported function (index 0): index of the corresponding type in the type list above

07 **<- Export list : Section list for exporting stuffs to JS-land**
0A(length in bytes) **01**(number of exported items)
06(string length) **74**(t) **65**(e) **73**(s) **74**(t) **49**(I) **66**(f) "testIf"
00 export category : function
00 function index in the function list above (indice 0)

0A **<- Code list : Section list for defining function bodies**
0F(length in bytes) **01**(number of code sections)
0D length in byte of the code section below
00 number of local variables other than arguments (none here)
20(get_local) **00** push the 8-byte float upon the stack
AA(trunc_s) pop the last stack value and push upon the stack its casting into a 4-byte integer.
04(If) ***7F****(returned type)* tested with the value upon the stack and notify that its returns a 4-float integer
41 **01** **If** condition is not null, push 1 upon the stack
05(Else) **Else** if condition is null …(not mandatory if no instructions in this case)
41 **00** push the returned value zero upon the stack
0B notify end of the If/Else the code
0B notify the end of the body code as usual`;

Binary translation is straight-forward; the only subtleties are :

- if/else code block must end with a **0B** token (like a function body);
- if no returned value type (ex: **7F**, equivalent in WAT to **(result i32)**) is specified for a **if** statement or if there's no instruction in the **if** clause, a **40** binary token must nonetheless directly follows the **if** token;

if	***04****(If)* ***40****(void)*
else	***05****(Else)*
i32.const 0	***41*** **00**
end	***0B***

- **else** token is optional (**05** can be omitted).

3.2.3. Code summary

Test the code as usual by pasting the code below into your browser console :

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;

const wasmIfElseExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   06(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   01       7C          number of function arguments and types
   01       7F          number of returned value and type

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first not-imported function (index 0): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   0A(length in bytes) 01(number of exported functions)
      06(string length) 74(t) 65(e) 73(s) 74(t) 49(I) 66(f)                    “testIf”
      00    export category : function
      00    function index in the function list above (indice 0)

0A          <- Code list : Section list for defining function bodies
   0F(length in bytes) 01(number of code sections)
      0D    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      20(get_local) 00          push the 8-byte float upon the stack
      AA(trunc_s)               pop the last stack value and push upon the stack its casting into a 4-byte integer.
      04(If) 7F(returned type)  IF tested with the value upon the stack and notify that its returns a 4-float integer
       41 01                    If condition is not null, push the returned value upon the stack (1)
      05(Else)                  Else if condition is null ...
       41 00                    push the returned value zero upon the stack
      0B                        notify end of the If/Else the code
      0B                        notify the end of the body code as usual
`;
let { instance: { exports: { testIf } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmIfElseExample)
);
window.alert(`Wasm - If/Else : testIf function with argument 10 yields ${testIf(10)}.
testIf function with argument 0 yields ${testIf(0)}.`);
})();
```

Note : the **i32.trunc_s/X** *operator was used so that you can learn it. However, it's not required if your condition is passed as an argument : as shown in the next chapter, you simply need to have an* ***i32*** *type in the signature, and the browser will truncate the argument you passed from javascript (ex: 0.4 becomes 0).*

3.3. Select

Writing complex code involves branchings, detailed in the next chapter. However if a logical condition only changes a value, one can simply use the **select** WAT operator. The dummy function written here is the equivalent of the javascript ternary operator (.. **?** .. **:** ..).

3.3.1. WAT code

```
(module
        (type                   ;; type  with index 0 in type list
                (func (param i32 i32 i32) (result i32))
        )
        (func (type 0)          ;; function with index 0 in function list and signature with index 0 in type list
                get_local 1     ;; value to be returned if non-null
                get_local 2     ;; value to be returned otherwise
                get_local 0     ;; condition for the select
                select          ;; result pushed upon the stack
        )
        (export "testTernary" (func 0))
)
```

3.3.2. Binary code / code summary

Binary token for **select** is **1B**. Paste the code below in your console for testing.

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmTernaryExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   08(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   03       7F 7F 7F  number of function arguments and types
   01       7F        number of returned value and type

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first not-imported function (index 0): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   0F(length in bytes) 01(number of exported items)
      0B(string length) 74(t) 65(e) 73(s) 74(t) 54(T) 65(e) 72(r) 6E(n) 61(a) 72(r)  79(y)                  “testTernary”
      00    export category : function
      00    function index in the function list above (indice 0)

0A          <- Code list : Section list for defining function bodies
   0B(length in bytes) 01(number of code sections)
      09    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      20(get_local) 01
      20(get_local) 02
      20(get_local) 00          ;; value at the top of the stack used as the condition
      1B(select)
      0B
`;
let { instance: { exports: { testTernary } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmTernaryExample)
);
window.alert(`Wasm - ternary : testTernary function with arguments 0, 2, 3 yields ${testTernary(0, 2, 3)}.
testTernary function with argument 1, 2, 3 yields ${testTernary(1, 2, 3)}.`);
})();
```

3.4. Branching

3.4.1. Principles

Control flow is mostly implemented in Wasm by using *branching* operators, based on the concept of *nested code blocks*. Inside a Wasm function body, various nesting levels are defined through code blocks. 3 code block types exist (.... represent instructions) :

- **if** .... (**else**) .... **end** code block (last chapter);
- **loop** .... **end** code block (next chapter);
- **block** .... **end** code block : the basic one, just here for the branching's sake.

Whenever a branching instruction with an indice (ex: **br 0**) is met at runtime, the program usually jumps from one code block to the end of another's (exception: **loop** .... **end** code block). The place where code execution is resumed after the jump ("branching") is called a *label*. Labels are implicit and are at the end of a code block - so that branching resumes code execution right after - except for the **loop** .... **end** code block where the label is at the start (so after a branching a loop instruction can be restarted, see chapter 3.6.). A branching instruction must be followed by an index which specifies if one wants to go to the nearest label (if **0**) or to an outermost one (the higher the value, the outermost). So the index is relative to the block it's used in. Let's clarify this with examples :

Branching to the current code block (index = 0) :

```
(type
        (func (param i32) (result f64))
)
(func (type 0)
        get_local 0                 ;; argument pushed upon the stack
        if (result f64)             ;; if argument is non-null (block must returns a f64 result)
                i32.const 1         ;; push a non-null i32 value to execute the inner if
                if (result f64)
                        f64.const 42        ;; arbitrary value required (returned value for if/else)
                        br 0                ;; go to the end of the current block
                        f64.neg             ;; never executed: negates the value upon the stack
                else
                        f64.const 0
                end         ;; implicit label for the inner if/else block : execution is resumed here!
                f64.const 2
                f64.mul             ;; multiply 42 by 2
        else
                f64.const 0
        end                         ;; implicit label for the topt if/else block
)
```

Given a non-null argument, this function returns 84, or -84 without the **br 0** statement.

Branching to the first enclosing outer block (index = 1) :

```
(type
        (func (param i32) (result f64))
)
(func (type 0)
        get_local 0
        if (result f64)
                i32.const 1          ;; push a non-null i32 value to execute the inner if
                if (result f64)
                        f64.const 42        ;; arbitrary because required (returned value for if/else)
                        br 1                ;; go to the end of the first outer block
                        f64.neg             ;; never executed: negates the value upon the stack
                else
                        f64.const 0
                end                 ;; implicit label for the inner if/else block
                f64.const 2         ;; never executed
                f64.mul             ;; never executed
        else
                f64.const 0
        end                 ;; implicit label for the outermost if/else block : execution  is resumed here!
)
```

Given a non-null argument, this function yields 42, or -84 without the **br 1** statement.

If you want to directly exit a deeply nested code block to go back to the main function, you can use **return** instead of **br X**.

Branching to the outermost label :

```
(type
        (func (param i32) (result f64))
)
(func (type 0)
        get_local 0
        if (result f64)
                i32.const 1          ;; push a non-null i32 value to execute the following if
                if (result f64)
                        i32.const 1          ;; push a non-null i32 value to execute the following if
                        if (result f64)
                                f64.const 42        ;; arbitrary returned value required for  if/else
                                return              ;; here equivalent to br 2
                                f64.neg             ;; never executed
                        else
                                f64.const 0
                        end                 ;; implicit label for the inner if/else block (if br 0)
                        f64.const 2         ;; never executed
                        f64.mul             ;; never executed
                else
                        f64.const 0
                end                 ;; implicit label for the intermediate if/else block (if br 1)
                f64.const 2         ;; never executed
                f64.mul             ;; never executed
        else
                f64.const 0
        end                 ;; implicit label for the outermost  if/else block : execution  is resumed
here!
)
```

Given a non-null argument, this function yields :

- 42 with **return** or **br 2**;
- 84 with **br 1** instead of **return**;
- 168 with **br 0** instead of **return**;
- -168 without **return**.

The **br_if** binary operator is a shortcut if you want to execute a branching right after a **if** ; it consumes the top stack value as a condition one, and branch accordingly :

```
(type
        (func (param i32) (result i32))
)
(func (type 0)
        block (result i32)
                i32.const 4
                get_local 0             ;; the condition is the function argument
                br_if 0                 ;; jump at the current block label (index 0) if non null argument
                drop                    ;; discard the value upon the stack (would be 4)…
                                        ;; … never executed if function argument is different from 0
                i32.const 2             ;; never executed if function argument is different from 0
        end                             ;; code execution resumed here if argument is different from 0
)
```

In the example above, one uses a **block ... end** code, block whose only role is to define a label. As for **if ... else ... end** block, the type of the value it pushes upon the stack) must be specified at its start (**(result i32)**). The dummy function above returns 4 if the argument is not null, 2 otherwise (last value upon the stack, 2, is dropped and replaced by 2)

You can combine multiple blocks and branchings operators for instance like this, to return a squared input value when the condition is true, and a negative one otherwise :

```
(type
        (func (param i32 f64) (result f64))     ;; input value is the second argument
)
(func (type 0)
        block (result f64)
                block (result f64)
                        get_local 1             ;; argument representing the processed value
                        get_local 0             ;; argument representing the condition
                        br_if 0                 ;; branch to current label if condition is true
                        f64.neg                 ;; executed if condition if false
                        br 1                    ;; ...then directly go to the end
                end     ;; code execution resumed here if br_if 0 branching
                f64.sqrt ;; code executed if condition is true
        end             ;; code execution resumed here if br 1 branching (condition false)
)
```

Notes :

- in this case it's better to simply use if/else;
- *you can use the **drop** WAT operator (binary token: **1A**) if you want to discard the last stack value - if not needed - from the Wasm stack while studying branching.*
  *As an example **i32.const 3** **drop** **i32.const 2** is equivalent to **i32.const 2***
- *You can also use the **nop** WAT operator (binary token: **01**) if you want to explicitly state "perform no operation" although a operator is required by the statement.*
- *You can use named references syntactic sugar for the label index as usual :*

```
block $B0 (result f64)
        block $B1 (result f64)
                ....
                br $B0
        end
end     ;; code execution resumed here
```

- *If a condition is forbidden, you can use the **unreachable** WAT operator (binary token: **00**) to stop the execution, by notifying the host (browser) that something illegal has just happened.*

3.4.2. Code summary

WAT token	i32.const	f64 type	block	br_if	(if nothing is returned by a block)	br	f64.neg	f64.sqrt	end
Binary token	*41*	*7C*	*02*	*0D*	*40*	*0C*	*9A*	*9F*	*0B*

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmBreaksExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   07(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   02       7F 7C       number of function arguments and types
   01       7C          number of returned value and type

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first not-imported function (index 0): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   0E(length in bytes) 01(number of exported functions)
      0A(string length) 74(t) 65(e) 73(s) 74(t) 42(B) 72(r) 65(e) 61(a) 6B(k) 73(s)          “testBreaks”
      00    export category : function
      00    function index in the function list above (indice 0)

0A          <- Code list : Section list for defining function bodies
   14(length in bytes) 01(number of code sections)
      12    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      02(block) 7C    open a block (parent) and specify its returned type (binary hex code forty if no returned value)
      02(block) 7C    open an inner block and specify its returned type (binary hex code forty if no returned value)
      20(get_local) 01          ;; value to be processed
      20(get_local) 00          ;; value at the top of the stack used as the condition
      0D(br_if) 00              ;; jump to the end of the current block if condition is met
      9A(negate)                ;; negate the value if condition is not met...
      0C 01(label index)        ;; .. then jump to the end of the parent block to avoid the code in between
      0B(end)                   ;; end of inner block
      9F(square root)           ;; square the value if condition is met...
      0B(end)                   ;; end of parent block
      0B                        ;; usual end of the function body
`;
let { instance: { exports: { testBreaks } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmBreaksExample)
);
window.alert(`Wasm - Break : testBreaks function with arguments 0, 42 yields ${testBreaks(0, 42).toFixed(6)}.
testBreaks function with argument 1, 42 yields ${testBreaks(1, 42).toFixed(6)} (square root of 42).`);
})();
```

To test this code, paste it in the browser console.

3.5. Indirect branching

3.5.1. Principle

Indirect branching was not introduced in the last chapter to keep it digestible. It relies on the same structure (nested code blocks with implicit code blocks) but allows to choose at runtime the branching index (ex: value right after **br** or **br_if**), via the **br_table** instruction.

To do so, the **br_table** operator must be followed by a list of label indices - which can be very long. **br_table** pops only one value from the stack, which determines the chosen index for the branching as if the list was an array and the popped value its index.
For instance :

```
br_table 2 1 3 0     ;; 2 1 3 0 corresponds to the the list of label indices
```

Then, the behavior depends on the last stack value, in the case above like this :

Last stack value	0	1	2	3
Equivalence	**br 2**	**br 1**	**br 3**	**br 0**

The following illustrative function performs more or less multiplications depending on the integer provided as an argument.

3.5.2. WAT code

```
(type
        (func (param i32) (result i32))
)
(func (type 0)
        block (result i32)
                block (result i32)
                        block (result i32)
                                block (result i32)
                                        i32.const 1         ;; default returned value
                                        get_local 0         ;; argument used as the list index
                                        br_table 2 1 3 0    ;; branching with label index list
                                        i32.const -1        ;; never executed
                                        i32.mul             ;; never executed
                                end                         ;; jump if argument = 3 (br 0)
                                i32.const 2
                                i32.mul
                        end                                 ;; jump if argument = 1 (br 1)
                        i32.const 3
                        i32.mul
                end                                         ;; jump if argument = 0 (br 2)
                i32.const 4
                i32.mul
        end                                                 ;; jump if argument = 2 (br 3)
        i32.const 5
        i32.mul
)
```

3.5.3. Binary code / code summary

WAT token	i32.const	i32 type	i32.mul	block	br_table	end
Binary token	*41*	*7F*	*6C*	*02*	*0E*	*0B*

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmIndBranchExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   06(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   01       7F          number of function arguments and types
   01       7F          number of returned value and type

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first not-imported function (index 0): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   11(length in bytes) 01(number of exported items)
      0D(string length) 74(t) 65(e) 73(s) 74(t) 49(I) 6E(n) 64(d) 42(B) 72(r) 61(a) 6E(n) 63(c)  68(h)       "testIndBranch"
      00    export category : function
      00    function index in the function list above (indice 0)

0A          <- Code list : Section list for defining function bodies
   29(length in bytes) 01(number of code sections)
      27    length in byte of the code section below
      00    number of local variables other than arguments (none here)
      02(block) 7F
       02(block) 7F
        02(block) 7F
         02(block) 7F
            41 01                   at least one value must be upon the stack
            20(get_local) 00        value at the top of the stack used as the index of the list below
            0E(br_table)
             03                     maximum index of the br_table list below
             02 01 03 00            LIST for br_table  :   2 1 3 0
            41 7F(-1 in lebEncoding)        ;; never executed - push minus one upon the stack
            6C                      never executed - multiplication
         0B(end)                    code resumed here if br_table equivalent to br 0
            41 02
            6C                      multiplication
         0B(end)                    code resumed here if br_table equivalent to br 1
            41 03
            6C                      multiplication
        0B(end)                     code resumed here if br_table equivalent to br 2
            41 04
            6C                      multiplication
       0B                           code resumed here if br_table equivalent to br 3
            41 05
            6C                      multiplication
      0B                            usual end of the function body
`;
let { instance: { exports: { testIndBranch } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmIndBranchExample)
);
window.alert(`Wasm - Branch : testIndBranch function with arguments 0, 1, 2, 3 returns ${testIndBranch(0)},
${testIndBranch(1)}, ${testIndBranch(2)}, ${testIndBranch(3)}, respectively.`);
})();
```

To test this code, paste it in the browser console.

3.6. Loops

3.6.1. Principles

The loop code block **loop ... end** is very peculiar because :

- it requires branching instructions for performing a usual loop;
- its label is at its beginning (instead of at its **end** for all the other code block types).

The typical pattern is as follows (pseudo WAT code) :

```
block (returned type)
        loop (returned type) ;; block label : code execution is resumed here after a br 0 inside this block
                ….      ;; loop instructions
                br_if 1 ;; exit the loop code block if the last value upon the stack (i32) is not null...
                br 0    ;; … otherwise keep looping
        end             ;; end of the loop block (code execution is never resumed here!)
end                     ;; the code execution is resumed here a
```

The function in this chapter will return the n^{th} power (2^{nd} argument) of a value (1^{st} argument)

3.6.2. WAT code

The code introduces a **tee** operation, which acts on local/global variables as a **set** operation followed by a **get** one, in order to save a value while keeping it on the top of the stack.

```
(type
        (func (param f64  i32) (result f64))    ;; local 0 (multiplier) and  local 1 (power)
)
(func (type 0) (local f64 i32)  ;; 2 extra local variables : local 2 (result) and local 3 (loop counter)
        get_local 0             ;; push the value to be multiplied by itself upon the stack
        set_local 2             ;; save variable upon the stack into the 3rd local (1st extra local)
                                ;; important : default value for an extra local is 0
        block                   ;; code blocks return no value (value popped after the last end)
                loop                            ;; br 0 from the loop leads us here
                        get_local 3             ;; get the loop counter (initial value: 0)
                        i32.const 1             ;; increment by one …
                        i32.add                 ;; … the loop counter

                        tee_local 3             ;; … and save it while keeping it upon the stack
                        get_local 1             ;; get the power
                        i32.ge_u                ;; loop exit if loop counter ≥ power
                        br_if       1           ;; in this case branching to the outer block
                                                ;; ...otherwise new iteration
                        get_local 0             ;; .. push the multiplier upon the stack
                        get_local 2             ;; ...push the temporary result upon the stack
                        f64.mul                 ;; multiply anew
                        set_local 2             ;;save the value upon the stack into the temporary result

                        br 0                    ;; keep looping : branching to the start of the loop
                end                             ;; nothing ever ends here for a loop … block !
        end                                     ;; the loop br 1 leads us here
        get_local 2                             ;; put the result upon the stack as the returned value
)
```

3.6.3. Binary code / code summary

This code also introduces extra local variables, declared like this to avoid repetitions :

Instead of ~~7F 7F 7C 7C 7C~~ you must write **02**(number of i32) **7F** **03**(number of f64) **7C**

WAT token	loop	i32.add	block	f64.mul	end	br	br_if	i32.ge_u	get_local	set_local	tee_local
Binary token	*03*	*6A*	*02*	*A2*	*0B*	*0C*	*0D*	*4F*	*20*	*21*	*22*

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmLoopExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   07(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   02       7C 7F      first argument is value to be multiplied, second is the power.
   01       7C         number of returned value and type

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first non-imported function (index 0): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   0C(length in bytes) 01(number of exported functions)
      08(string length) 74(t) 65(e) 73(s) 74(t) 4C(L) 6F(o) 6F(o) 70(p)          "testLoop"
      00    export category : function
      00    function index in the function list above (indice 0)

0A          <- Code list : Section list for defining function bodies
   29(length in bytes) 01(number of code sections)
      27    length in byte of the code section below
      02                  <- Total number of extra local variables
      01                  number of extra local variables in the first contiguous group (same type)
        7C                type of the group variable(s)
      01                  number of extra local variable in the second contiguous group (same type)
        7F                      type of the group variable(s)
      20(get_local) 00
      21(set_local) 02    save the multiplier in the 3rd local (= first extra local variable)
      02(block) 40(void)  parent code block, returns nothing (void)
       03(loop) 40(void)  code is resumed here (implicit label) after a br 0 inside
        20(get_local) 03  push the loop counter upon the stack (4th local = second extra local variable)
        41 01             push 1 upon the stack
        6A                increment the loop counter
        22(tee_local) 03  save the loop counter without changing the stack
        20(get_local) 01  get the power (second local/argument)
        4F(>=)            push 1 upon the stack if loop counter >= power, 0 otherwise
        0D(br_if) 01      if last value is non-null, exiting the loop by branching to the parent label
        20(get_local) 00  ...otherwise multiply the current result by the multiplier...
        20(get_local) 02
        A2(multiply)
        21(set_local) 02  ...and save it
        0C(br) 00         ... then repeat the loop (br 0 goes to the implicit label at the top of the loop)
       0B(end)            code is never resumed here after a br 0 inside a loop ... end!
      0B(end)             parent label - code resumed here if br_if 0 (equivalent to a javascript break)
     20(get_local) 02     push the 3rd local (first extra local variable) containing the final result upon the stack
      0B                  ;; usual end of the function body`;
let { instance: { exports: { testLoop } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmLoopExample));
window.alert(`Wasm - Loop : testLoop function with 3 as first argument and 2nd argument being respectively
equal to 2, 5, 200 returns ${testLoop(3, 2)},  ${testLoop(3, 5)} and ${testLoop(3, 200)}.`);
})();
```

To test this code, paste it in the browser console.

3.7. Indirect function calls via table and element

Sometime you want to choose the function to be called at runtime. For security reason, Wasm enforces you to define such functions inside a structure called *table*, where each function is referenced by an index. A *table section* has the identifier with value **04**.
As for the memory section, limited to one per Wasm module, Wasm 1.0 is restricted to a single table section. Various ways can be used to create Wasm tables, by either :

1. creating a Wasm table object in javascript, then importing it;
2. or defining a table section directly inside a Wasm module.

We will start with the 1st method, because the 2nd one is more complex as one has to introduce an extra section named *element*, and the module is less lisible because bigger. The example in the following chapters consists in choosing at runtime a binary arithmetical operation via an index given as an argument: 0 for an addition, 1 for calculating the minimum of two arguments, and 2 for calculating the maximum instead.

3.7.1. Creating a table externally

3.7.1.1. Setting up a table

The logic to create a table is similar to the one for creating global or memory objects (chapter 2.2.1.1.), with a constructor accepting a descriptor argument, and the new Wasm object being imported into the Wasm module as a 2-level **wasmImport** object :

```
const tableDescriptor = {
   initial: 3,              // number of function to be indexed in the table
   maximum: 3,              // optional, here if we are sure that we only need 3 functions
   element: 'anyfunc'       // mandatory, only possibility for Wasm 1.0
};
let funcs = new WebAssembly.Table(tableDescriptor);
// Associating indices to functions
funcs.set(0, /* first function to be called indirectly goes here */ );
funcs.set(1, /* 2nd function to be called indirectly goes here */ );
funcs.set(2, /* 3rd function to be called indirectly goes here */ );

let wasmImport = { indirect: { funcs } };
let { instance } = await WebAssembly.instantiate(wasmBinary, wasmImport);
```

The important points are that :

- table functions must be as defined as **'anyfunc'** in the **element** field of their descriptor (more categories in the future versions of Wasm);
- the only accepted values for the table setter (**set**) are Wasm functions (thus exported from another module) or **null**.

Note: a Wasm table object can be grown at runtime with the **grow** *method (like Wasm memory!)*

Hence, we need two modules :

- The main one, using the table;
- The one defining and exporting the required functions for setting up the table in javascript.

3.7.1.2. WAT code

The code for the module exporting Wasm functions to fill our Wasm table object is :

```
(module
    (type
        (func (param f64 f64) (result f64))
    )
    (func (type 0)
        get_local 0
        get_local 1
        f64.add
    )
    (func (type 0)
        get_local 0
        get_local 1
        f64.min
    )
    (func (type 0)
        get_local 0
        get_local 1
        f64.max
    )
    (export "add" (func 0))
    (export "min" (func 1))
    (export "max" (func 2))
)
```

The code for our main module is :

```
(module
    (type     ;; index 0 type for the table functions
        (func (param f64) (param f64) (result f64))
    )
    (type     ;; index 1 type for the "main" function : 3rd argument is for selecting the table function via its index
        (func (param f64) (param f64) (param i32) (result f64))
    )
    (import "indirect" "funcs"
        ;; implicitly create a table section, with index 0
        (table  3  3 anyfunc)        ;; first number is initial size, 2nd one (optional) is max. size,
                                     ;; ... and the anyfunc (or funcref) string means
                                     ;; ... that all function signatures are accepted.
    )
    (func (type 1)
        get_local 0          ;; first two arguments are to be used by table functions
        get_local 1
        get_local 2                  ;; 3rd argument is the table index for indirect call
        call_indirect (type 0)       ;; calling the table function whose index is upon the stack
                                     ;; the type must be specified right after call_indirect!
    )
    (export "testTable" (func 0))
)
```

Here the table is imported as a global, function or memory section. The table node has 2 numbers (limits) has a memory : the first one is the initial size, the second one is the maximum.

3.7.1.3. Binary code / code summary

```
(async () => {
// Auxiliary module (for the external functions)
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmAuxiliaryModule = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
  07(length in bytes) 01(number of defined types : number of types sections)
  60      define a function type for type with index 0
  02      7C 7C
  01      7C          number of returned value and type
03          <- Function list : Section list for defining functions
  04(length in bytes) 03(number of defined functions)
  00      first non-imported function (index 0): index of the corresponding type in the type list above
  00      2nd non-imported function (index 1): index of the corresponding type in the type list above
  00      3rd non-imported function (index 2): index of the corresponding type in the type list above
07          <- Export list : Section list for exporting stuffs to JS-land
  13(length in bytes) 03(number of exported items)
    03(string length) 61(a) 64(d) 64(d)                          "add"
    00    export category : function
    00    function index in the function list above (indice 0)
    03(string length) 6D(m) 69(i) 6E(n)                          "min"
    00    export category : function
    01    function index in the function list above (indice 1)
    03(string length) 6D(m) 61(a) 78(x)                          "max"
    00    export category : function
    02    function index in the function list above (indice 2)
0A          <- Code list : Section list for defining function bodies
  19(length in bytes) 03(number of code sections)
        // Add
    07    length in byte of the code section below for the first section in the function list
    00    number of local variables other than arguments (none here)
    20(get_local) 00
    20(get_local) 01
    A0(add)                 8-byte float addition
    0B                      end of the function body
        // Min
    07    length in byte of the code section below for the first section in the function list
    00    number of local variables other than arguments (none here)
    20(get_local) 00
    20(get_local) 01
    A4(min)                 return the minimum of the 2 8-byte floats above
    0B                      end of the function body
        // Max
    07    length in byte of the code section below for the first section in the function list
    00    number of local variables other than arguments (none here)
    20(get_local) 00
    20(get_local) 01
    A5(max)                 return the maximum of the 2 8-byte floats above
    0B                      end of the function body`;
const { instance: { exports: { add, min, max } } } = await WebAssembly.instantiate(
  wasmConvertDirtyHexToBinary(wasmAuxiliaryModule)
);
// Code to create the table from Javascript
const tableDescriptor = {
  initial: 3,               // number of function to be indexed in the table
  maximum: 3,               // optional, here if we are sure that we only need 3 functions
  element: 'anyfunc'        // mandatory, only possibility for Wasm 1.0 (anyfunc==funcref in the spec)
};
const  funcs = new WebAssembly.Table(tableDescriptor);
funcs.set(0, add);
funcs.set(1, min);
funcs.set(2, max);
let wasmImport = { indirect: { funcs } };
```

```
// Main module (the one using the table)
    const wasmTable = `${wasmHeader}
    01          <- Type list : Section list for defining types (parameters and results)
      0E(length in bytes) 02(number of defined types : number of types sections)
      60        define a function type for type with index 0 (table functions)
      02        7C 7C
      01        7C                  number of returned value and type

      60        define a function type for type with index 1 (“testTable” function)
      03        7C 7C 7F
      01        7C                  number of returned value and type

    02          <- Import  list : Section list for importing items from javascript
      15(length in bytes) 01(number of defined types : number of types sections)
      08(length in bytes) 69(i) 6E(n) 64(d) 69(i) 72(r) 65(e) 63(c) 74(t)       import “namespace” : “indirect”
      05(length in bytes) 66(f) 75(u) 6E(n) 63(c) 73(s)                        imported variable: “funcs”
      01        import category : define a table import (category id : 0 for function, 1 for table, 2 for memory, 3 for global)
      70        table elements are of type anyfunc (= funcref in the spec)
      01        like for memory limits: 1 = maximum is defined, 0 = no maximum
      03        initial table size
      03        maximum table size

    03          <- Function list : Section list for defining functions
      02(length in bytes) 01(number of defined functions)
      01        first non-imported function (index 0): index of the corresponding type in the type list above

    07          <- Export list : Section list for exporting stuffs to JS-land
      0D(length in bytes) 01(number of exported items)
        09(string length) 74(t) 65(e) 73(s) 74(t) 54(T) 61(a) 62(b) 6C(l) 65(e)        “testTable”
        00      export category : function
        00      function index in the function list above (index 0)

    0A          <- Code list : Section list for defining function bodies
      0D(length in bytes) 01(number of code sections)
        0B      length in byte of the code section below for the first section in the function list
        00      number of local variables other than arguments (none here)
        20(get_local) 00
        20(get_local) 01
        20(get_local) 02            index of the indirectly-called function in the function
indirect function call below using the last stack value as the index of the called function in the table
        11(call_indirect) 00(function type) 00(reference to the table section : always 0 for Wasm 1.0!)
        0B                          end of the function body`;
    const { instance: { exports: { testTable } } } = await WebAssembly.instantiate(
      wasmConvertDirtyHexToBinary(wasmTable),
      wasmImport
    );
    window.alert(`Wasm - Table : testTable function with 2.5 and 10 as first arguments, and respectively 0, 1, 2 as
    3rd argument returns ${testTable(2.5, 10, 0)} (add),  ${testTable(2.5, 10, 1)} (min) and ${testTable(2.5, 10, 2)}
    (max).`);
    })();
```

Test the code as usual by pasting the code of the 2 last pages into your browser console.

3.7.2. Creating a table inside a Wasm module

In the case where you want to put everything into a single Wasm module, you have to use an extra sections called *element* section to *initialize* your table section. *Element sections* have the identifier with value 09.

Note : the order of the function in the table have been changed to avoid misunderstanding between index roles (first is maximum, then addition, and finally minimum)

3.7.2.1. WAT code

The logic for an *element section* is the same as the one for a *data section* (see chapter 2.2.3.6.) :

- an offset - the starting index in the table where the initialization begins - specified as an instruction (**i32.const** *offset*);
- a space-separated list of values, this time for each index.

```
(module
        (type       ;; index 0 type for the table functions
                    (func (param f64) (param f64) (result f64))
        )
        (type       ;; index 1 type for the "main" function : 3rd argument is for selecting the table function via its index
                    (func (param f64) (param f64) (param i32) (result f64))
        )
        (table
                    3           3 anyfunc       ;; 1st number is initial size, 2nd one (optional) is max. size,
                                                ;; 3rd one is element type (always anyfunc)
        )
        (elem
                    (i32.const 0) 2 0 1         ;; i32.const 0 means that the table is initialized from its first index
                                                ;; space-separated values 2 0 1 are indices to functions (below) ...
                                                ;; ...corresponding respectively to table indices 0, 1 and 2.
        )
        (func (type 0)                          ;; function with index 0
                    get_local 0
                    get_local 1
                    f64.add
        )
        (func (type 0)                          ;; function with index 1
                    get_local 0
                    get_local 1
                    f64.min
        )
        (func (type 0)                          ;; function with index 2
                    get_local 0
                    get_local 1
                    f64.max
        )
        (func (type 1)                          ;; function with index 3
                    get_local 0         ;; first two arguments are to be used by table functions
                    get_local 1
                    get_local 2                 ;; 3rd argument is the table index for indirect call
                    call_indirect (type 0)      ;; calling the table function whose index is upon the stack
                                                ;; the type must be specified right after call_indirect!
        )
        (export "testTable" (func 3))
)
```

3.7.2.2. Binary code / code summary

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
};          const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmTable2 = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   0E(length in bytes) 02(number of defined types : number of types sections)
   60       define a function type for type with index 0 (table functions)
   02       7C 7C
   01       7C                    number of returned value and type

   60       define a function type for type with index 1 ("testTable" function)
   03       7C 7C 7F
   01       7C                    number of returned value and type

03          <- Function list : Section list for defining functions
   05(length in bytes) 04(number of defined functions)
   00       first non-imported function (index 0): index of the corresponding type in the type list above
   00       2nd non-imported function (index 1)
   00       3rd non-imported function (index 2)
   01       4th non-imported function (index 3)

04          <- Table list : Section list for storing Wasm functions for indirect calls
   05(length in bytes) 01(number of defined tables)
   70       table elements are of type anyfunc (= funcref in the spec)
   01       like for memory limits: 1 = maximum is defined, 0 = no maximum
   03       initial table size
   03       maximum table size

07          <- Export list : Section list for exporting stuffs to JS-land
   0D(length in bytes) 01(number of exported items)
      09(string length) 74(t) 65(e) 73(s) 74(t) 54(T) 61(a) 62(b) 6C(l) 65(e)                  "testTable"
      00(export category : function)        03(function index in the function list above)

09          <- Element list : Section list for initializing tables
   09(length in bytes) 01(number of exported items)
      00                          index of the initialized table (always 0 for Wasm 1.0!)
      41(const) 00                defining with an instruction the table index at which initialization is starting
      0B                          end of block to specify that the instructions are over
      03(list length)   02 00 01  list of values specifying the function indices for the 3 first table indices

0A          <- Code list : Section list for defining function bodies
   25(length in bytes) 04(number of code sections)

      07(section length in bytes) 00(number of extra local variables)
      20(get_local) 00                  20(get_local) 01
      A0(add)                     8-byte float addition
      0B                          end of the function body

      07(section length in bytes) 00(number of extra local variables)
      20(get_local) 00                  20(get_local) 01
      A4(min)                     return the minimum of the 2 8-byte floats above
      0B                          end of the function body

      07(section length in bytes) 00(number of extra local variables)
      20(get_local) 00                  20(get_local) 01
      A5(max)                     return the maximum of the 2 8-byte floats above
      0B                          end of the function body

      0B(section length in bytes) 00(number of extra local variables)
      20(get_local) 00                  20(get_local) 01
      20(get_local) 02            index of the indirectly-called function in the function
indirect function call below using the last stack value as the index of the called function in the table
      11(call_indirect) 00(function type) 00(reference to the table section : always 0 for Wasm 1.0!)
      0B                          end of the function body`;
const { instance: { exports: { testTable } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmTable2) );
window.alert(`Wasm - Table : testTable function with 2.5 and 10 as first arguments, and respectively 0, 1, 2 as
3rd argument returns ${testTable(2.5, 10, 0)} (max),  ${testTable(2.5, 10, 1)} (add) and ${testTable(2.5, 10, 2)}
(min).`); })();
```

Test the code as usual by pasting the code above into your browser console.

3.8. Recursion without troubles

⚠️ Not fully integrated => Not for production. Use NodeJs (12.6) instead of the browser ⚠️

Usually you use loops when your task involve several iterations. Let's take a (too) typical example, involving the factorial (! symbol) numerical calculation : 5! = 5 * 4 * 3 * 2 * 1 = 120

```
function factorial(number=1) {
   let result = 1;
   for (let i = number; 1 < i; i--) {
      result *= i;
   }
   return result || 1;
}
```

The function above has no inherent drawbacks, but its code can be describe in a more intuitive way through recursion.

3.8.1. Principles

The principles of recursion consist in :

- finding a trivial base case : for instance factorial(X) = 1 for a integer X lower than 2
- discovering a useful relationship :
  by definition **factorial(number) = number * factorial(number - 1)** if **number - 1 >= 0**
- combining these 2 last steps for the actual implementation :

```
function recursiveFunction(value) {
   if (/* is base case? */) return trivialResult;
   return /* result of some operation with value and recursiveFunction(value - 1) */;
}
```

So with our example :

```
function factorialR(number=1) {
   if (number < 2) return 1;
   return number * factorialR(number - 1);
}
```

This code is elegant but can be troublesome, as during its execution the runtime has to keep the context of each call. Let's take an analogy for - hopefully - simplicity's sake : Say that you're a queen sending a mounted courier to discuss about diplomatic matters with a king in a far away country (task **factorialR(number)**):

- the travel can't be done at once, and for various reasons each part is done by a different courier : the first courier rides up to a specific hostel in the neighbouring country, where he meets another trusted courier and gives him/her the same, remaining task (**factorial(number - 1)**), and finally let some instructions in the hostel as a future task for the returning courier (**number** in **number * factorial(number - 1)**)
- And so the same process is repeated at a specific hostel in the next country ...

We can see a flaw here : the returning couriers require some instructions left by the incoming courier (**number** in **number * factorial(number - 1)**), so extra things must be kept by the

hostel directors, who bill for that the queen accordingly; the more countries, the more expensive.

These hostels are similar to the browser or NodeJs(V8) stack, which keeps the context of each function call, and is a limited resource. Solution? Instead of keeping some instruction in his/her destination hostel, a courier tells to the next one to keep with him/her personal instructions - *payload* - containing related tasks to be done at his/her return.

In a nutshell, the optimization consists in simply returning the executed recursive function - instead of returning an operation involving both the recursive function and extra stuffs : this is called *Tail Call Optimization* (TCO), and is a game changer as we don't stress the stack at runtime so we can as an example do huge factorial without hanging the system (aside the processing time), which would otherwise give this type of errors for Javascript:

```
Uncaught RangeError: Maximum call stack size exceeded
```

That for, the programming language and its runtime must support TCO, otherwise the platform will still use the stack even if we return the executed function invocation. As an example for javascript, *Tail Call Optimization* nearly ended up in ES6. You could have used the function below even for high numbers, where you transfer an additional payload to the next call :

```
function factorialTCO(number=1, payload=1) {
  if (number < 2) {
    return payload;
  } else {
    return factorialTCO(number - 1, payload * number);
  }
}
```

The payload of the initial courier must have no impact on the process. After all, the payload is only defined in our metaphor as a communication mean between "couriers", not as something required to communicate with the "queen" initiating the task. In the case of the factorial function, a no-impact payload is one.

But this book is about Wasm, so let's test this bleeding-edge feature on NodeJs.
The related proposal tells us to use the **return_call** instruction instead of the **call** one in order to recursively call a function the TCO way.

Note : you can use TCO with indirectly-called function by replacing ***return_call*** *by* ***return_call_indirect****, whose usage is similar to* ***call_indirect*** *(see last chapter).*

3.8.2. WAT code

```
(module
        (type ( func (param f64) (param f64) (result f64) ) )          ;; signature with index 0 - TCO function
        (type ( func (param f64) (result f64) ) )                      ;; signature with index 1 - factorial
        (func (type 0) (local f64)                          ;; TCO function
                get_local 0                     ;; factorial number
                f64.const 2
                f64.lt                          ;; is base case (number < 2)?
                if (result f64)
                        get_local 1             ;; returns payload
                else                            ;; recurse
                        get_local 0
                        get_local 1
                        f64.mul
                        set_local 2             ;; save new payload in extra local
                        get_local 0
                        f64.const 1
                        f64.sub                 ;; new number upon the stack
                        get_local 2             ;; new payload upon the stack
                        return_call 0           ;; TCO call
                end
        )
        (func (type 1)                  ;; wrapper to ease the use of the TCO factorial
                get_local 0             ;; first argument (number)
                f64.nearest             ;; round to nearest integer
                f64.const 1             ;; second argument (payload) equals to one by default
                call 0
        )
        (export "factorial" (func 1))
)
```

3.8.3. Binary code / code summary

Test recursion by installing the latest NodeJs version (12.6), creating a **tco.js** file whose content is at the next page, and executing : **node --expose-wasm --experimental-wasm-return_call tco.js**

WAT token	f64	f64.const	f64.mul	set_local	f64.sub	if	else	end	f64.lt	f64.nearest	return_call	return_call_indirect
Binary token	*7C*	*44*	*A2*	*21*	*A1*	*04*	*0D*	*0B*	*63*	*9E*	*12*	*13*

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;

function convertIEEE754toHexa(value) {
   const arr = [null, null, null, null, null, null, null, null];
   const ab = new ArrayBuffer(8);
   const dv = new DataView(ab); dv.setFloat64(0, value);
   for (let i = 0;  i < 8; i++) { arr[8 - i] = dv.getUint8(i).toString(16).padStart(2, '0').toUpperCase(); }
   return arr.join(' ');
}

const wasmRecursionExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   0C(length in bytes) 02(number of defined types : number of types sections)
   60       define a function type for type with index 0                    Tail-call function
   02       7C 7C    number of function arguments and types
   01       7C       number of returned value and type

   60       define a function type for type with index 0
   01       7C       number of function arguments and types
   01       7C       number of returned value and type

03          <- Function list : Section list for defining functions
   03(length in bytes) 02(number of defined functions)
   00       first not-imported function (index 0): index of the corresponding type in the type list above
   01       2nd not-imported function (index 1): index of the corresponding type in the type list above

07          <- Export list : Section list for exporting stuffs to JS-land
   0D(length in bytes) 01(number of exported functions)
      09(string length) 66(f) 61(a) 63(c) 74(t) 6F(o) 72(r) 69(i) 61(a) 6C(l)                    “factorial”
      00(export category : function) 01(function index in the function list above (indice 1))

0A          <- Code list : Section list for defining function bodies
   40(length in bytes) 02(number of code sections)
      2D    length in byte of the code section below - Tail call function
      01 01 7C       number of local variables other than arguments, defined by types
      20(get_local) 00          push the 8-byte float upon the stack
      44(const) ${convertIEEE754toHexa(2)}(two)   minimum number, for base case
      63(lower than)
        04(If) 7C(returned type) IF tested with the value upon the stack and notify that its returns a 8-float integer
        20(get_local) 01        push one upon the stack
        05(Else)                Else if condition is null ...
        20(get_local) 00        push number upon the stack
        20(get_local) 01        push payload upon the stack
        A2(mul)                 new payload = number * payload
        21(set_local) 02        save result - new payload - in extra local (index 2)
        20(get_local) 00        get number anew
        44(const) ${convertIEEE754toHexa(1)}(one)
        A1(sub)                 number - 1          1st argument for recursive call
        20(get_local) 02        new payload         2nd argument for recursive call
        12(return_call) 00      recursively calling this function (index 0) with TCO
      0B                        notify end of the If/Else the code
      0B                        notify the end of the body code as usual

      10 length in byte of the code section below - wrapper
      00 number of extra local variables
      20(get_local) 00          push the 8-byte float upon the stack
      9E(nearest)               round the input if decimal number entered by mistake
      44 ${convertIEEE754toHexa(1)}(one)          initial payload
      10(call) 00
      0B                        notify the end of the body code as usual
`;
let { instance: { exports: { factorial } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmRecursionExample)
);
console.log(`Wasm - Tail call optimisation : 150! equals ${factorial(150)}.`);
})();
```

4. Extra performance

4.1. Threads

4.1.1 Prerequisites

⚠️ Use the last version of Chrome/Chromium for this chapter ⚠️

4.1.1.1. Context

In order for several people - helpers - to work efficiently on several items :

1. There must be a sufficient number of items to be processed : otherwise the time required to give instructions, monitor the process then retrieve the “results” is as significant as the time taken to do it alone!
2. The tasks must be clearly defined to avoid interferences between people.

This is the same for adding more computing power on a specific tasks (Wasm functions), where the coordinator is the javascript program, and the helpers are Wasm instances encapsulated inside *Web Workers aka* real threads for the web :

1. The more data you have to work on, the more the time dedicated to managing each *worker* is low relative to its processing time;
2. The steps must be precise for fast and coordinated operations :
    a. The items (values to be processed by the functions) are typically accessed through *SharedArrayBuffers*, which are like *ArrayBuffers* but can be shared between *workers*. Hence, the processed values are stored into a *typed array* built upon the *SharedArrayBuffer* (see Appendix A.0.2.1. as a reminder).
    b. If the workers work on the same values (*concurrency,* as an example on the same indices of the array based on the *SharedArrayBuffer*), the operations must be coordinated to avoid *workers* getting on each other’s way, leading to bugs very difficult to deal with.

As of June 2019, only Chrome/Chromium and the last version of Edge support *SharedArraBuffers.* For the little story, this feature was available end 2017 in the last version of each major browser, but was disabled in early 2018 after the discovery of possible huge security issues (see Meltdown / Spectre).

Points 2.a. and 2.b. will be the focus of chapters 4.2. and 4.3., respectively.

But first of all, let’s introduce *workers* more concretely.

Notes :

- *Other types of workers do exist, but only standard workers and Service Workers are introduced in this book;*
- *Official documentation : https://github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md*

4.1.1.2. Web Worker basics

A *web worker* requires its own code - such as inside a **worker1.js** file hosted on a server :

```
const worker1 = new Worker('worker1.js');
```

Such threads communicate with the javascript program through *messages*, and take actions accordingly. Messages can be anything that can be cloned.

// From the main program :

```
worker1.onmessage = handleMessage;          // executed when worker1 sends something to the main program
const data = …;                             // data to be processed by worker1
worker1.postMessage(data);                  // data is sent to worker1 via cloning (may be expensive!)
function handleMessage(event) {
   const { data } = event;      // received result (see code below) contained in the data field of the message event
   ...
}
```

// Inside worker1.js :

```
self.onmessage = handleTask;          // executed when worker1 sends something (worker1.postMessage)
function handleTask(event) {
   const { data } = event;            // received data contained in the data field of the message event
   const result = …;                  // say variable result contains the result of the process
   postMessage(result);               // send the result back (trigger handleMessage in the code above)
}
```

To avoid the cost of cloning significant amount of data, you can either :

- use a *SharedArrayBuffer* as previously stated;
- or if you only need a result (the data can be disposed afterward), you can transfer the data ownership to the worker (= no cloning, just passing references => damn fast), by providing a second argument specifying a list of *transferable* values :

```
const ab = new ArrayBuffer(10);
worker1.postMessage(ab, [ab]);        // the data is ab, in the data only ab is transferable
```

or for an object :

```
const obj = {
   arr1: [1, 2, …, 1000],
   arr2: [2, 5, …, 2000],
   b: true
};
// the data is obj : obj.arr1 and obj.arr2 are transferred, while obj.b is cloned
worker1.postMessage(obj, [obj.arr1, obj.arr2]);
```

Notes :

- *You can only transfer certain specific objects, such as ArrayBuffer, ImageBitMaps...*
- *for the curious ones, please also check this great and fresh article!*

As *workers* require files, a minimalistic server is necessary. To avoid to deal with multiple files and to learn new stuffs in the process, we're going to host a single file for all the workers by intercepting request and forging responses from the frontend (chapter 4.1.4.2).

4.1.1.3. Architecture

The architecture is going to be as follows :

- [main program] : prepares a Wasm binary code whose functions can be executed on multiple threads, as an example by splitting the operations on different parts of a *SharedArrayBuffer* containing the values to be processed;
- [main program] : prepares the **imports** object - can be a partial one - and defines a thread number (workers);
- [main program] : compiles the Wasm binary code via **WebAssembly.compile**;
- [main program] : initializes the workers by creating them then sending them at least the compiled **module** and the **imports** object;
- [worker side] : instantiates the compiled Wasm **module** with its **imports** object in order to get the Wasm exported function (**exports**);
- [worker side] : notifies the main program when the worker is operational;
- [main program] : waits for all the workers to be ready to start a task
- [main program] : sends a task - Wasm function name and arguments
- [worker side] : receives a task, processes it then returns a result - even *undefined* to notify the end of the process (if the result is only a modified *SharedArrayBuffer*)
- [main program] : receives the worker results and the function execution is over.

Moreover, we will assume that each worker is strictly dedicated to Wasm related tasks, i.e. with no complex message handling : the messages and their answers are expected to be chronologically ordered - if the main program sends a message to a given worker, it expects that the related answer directly follows, and not after another message! Hence, each worker - considered independently of its Wasm instance - is expected to be not very smart and not to do tricky asynchronous stuffs.

4.1.1.4. Thread setup - worker side

4.1.1.4.1. Serving files from Chrome/Chromium

For simplicity's sake, let's use Chrome in this whole chapter, and install the Web Server for Chrome extension. Create a new folder (ex: "threads"), start the extension and choose this folder. Then, use localhost:8888 instead of www.blank.org as the default testing page.

4.1.1.4.2. Dynamically forging worker scripts

4.1.1.4.2.1. Using a Service Worker

Service Worker is not detailed here. See this awesome free course on Udemy.
But in a nutshell, a *Service Worker* is a supercharged *worker* that can act as a reverse proxy / small server inside your browser :

- It has its own life-cycle, decoupled from the main program;
- When it's activated - see the *Application* tab in your browser console - it can :
    - intercept HTTP requests and make some decisions about their processing;
    - save HTTP responses through the Cache API (static content) or IndexedDB (do yourself a favor and use the wrapper kindly provided by Mr Archibald);
    - deal with notifications like native apps …

That for, you just have to register it first in your main program as follows :

```
if ('serviceWorker' in navigator) {                    // check if Service Worker is supported
   const registration = await activateServiceWorker('/sw.js');          // load sw.js as a service worker
}

async function activateServiceWorker(url='', options={ scope: '/' }) {
      const registration = await navigator.serviceWorker.register(url, options);
      return Promise.resolve(registration);
}
```

Then, provided your server (ex: *Web Server for Chrome*) points to the folder with the **sw.js** file, you can write code in this file to dynamically generate HTTP responses.

Notes:

- *Not meaningful to check for Service Worker support, as evident in this chapter;*
- *from now on till the end of the sub-chapter, the code refers to the one inside **sw.js**.*

That for, you can intercept a request via a specific listener, and the callback function - **handleFetch** - must send its response through an **event.respondWith** function, accepting a Response Object (Response constructor) or a Promise resolving to a HTTP response (see Appendix A.0.2.7. for Promises if this sentence sounds strange) :

```
self.addEventListener('fetch', handleFetch);
function handleFetch(event) {
   const { request: { url } } = event;      // for the intercepted request, get the url
   …                                        // prepare the response accordingly
   event.respondWith(/* Response object or Response wrapped inside a Promise */);
}
```

In our case, we want to generate a HTTP response containing a script to build a new *worker* - **workerXX.js** - so we simply check the **url** via a regular expression (see final code in chapter 4.1.1.4.2.2.). If so, we have to return a Promise returning a Response containing the script :

```
function handleFetch(event) {
  const { request: { url } } = event;
  if ( /* is worker script url ? */ ) {
    const workerScript = /* script content for workerXX.js */
    const responseOptions = { headers: { 'Content-Type': 'application/javascript'} };
    event.respondWith(new Response(workerScript, responseOptions));
  }
}
```

Otherwise we simply fetch the response on our local server :

```
function handleFetch(event) {
  const { request: { url } } = event;
  if ( /* is worker script url ? */ ) {
  } else {
    event.respondWith( fetch(url) );
  }
}
```

Let's now focus ourselves on the dynamically-generated script used to create workers.

*Note: it would have been more direct to use the same script for all the workers (to simply use a **worker.js** file instead of **sw.js** one), and pass additional context via messages, but I find - personal opion - the current approach more convenient for tinkering.*

For starting Wasm functions, one requires a name - **funcName** - as expected in the **exports** field of the Wasm instance; and optional arguments **args** as an array. Then, the worker posts the result of the Wasm function execution :

```
const workerScript = `
  let isInit = false;              // true if the Wasm instance is ready
  let functions = null;            // will correspond to Wasm exports
  self.onmessage = async ({data}) => {          // payload in the data field of the event
    if (isInit && data.functionName) {
      if (data.args) {
        postMessage( functions[data.functionName](...data.args) );
      } else {
        postMessage( functions[data.functionName]() );
      }
    } else {
      ....
    }
  }`;
```

If the payload - **data** - doesn't contain a **functionName** field, this corresponds to the initialization step : the worker expects to receive a **module**, an **imports** object and additional context information such as the number of workers :

```
const workerId = …;                    // Inferred from the url
const workerScript = `
   let isInit = false;
   let functions = null;          // will correspond to Wasm exports
   self.onmessage = async ({data}) => {          // payload in the data field of the event
      if (isInit && data.functionName) {
         ….
      } else {
         try {

            // Prepare context information as Wasm Globals
            const id = new WebAssembly.Global({          // thread id
               value: 'i32',
               mutable: false
            }, ${workerId});

            const max = new WebAssembly.Global({          // max. number of threads
               value: 'i32',
               mutable: false
             }, data.threadNumber);

            const byteSize = data.bufferSize * 2**16;          // ** = ES6 exponent operator

            const size = new WebAssembly.Global({          // max. number of threads
               value: 'i32',
               mutable: false
             }, byteSize);

            // completing the imports object
            data.imports.thread = { id, max };
            data.imports.shared.size = size;

            ({ exports: functions } = await WebAssembly.instantiate(data.module, data.imports));
            isInit = true;
            postMessage('OK');
         } catch(error) {
         postMessage('FAILURE: ' + error );
      }
   }`;
```

*Note : Up to now, we've said that the **WebAssembly.instantiate** function accepts binary code (**ArrayBuffer** or **Uint8Array**) as its first arguments. Actually, this function has an overloaded version, used here, accepting a Wasm module as its first argument.*

4.1.1.4.2.2. Code summary

Copy-paste the code below into a new file - **sw.js** - into your new directory (chapter 4.1.1.4.1.) and make *Web Server for Chrome* points on this folder if this is not already the case.

```
// Intercept every HTTP request, execute a function to dynamically generate the code for a specific Web Worker
self.addEventListener('fetch', handleFetch);

const responseOptions = {                       // Header for workerXX.js
   status: 200,
   statusText: 'OK',
   headers: { 'Content-Type': 'application/javascript'}
};

function handleFetch(event) {
   const { request: { url } } = event;      // for the intercepted request, get the url
   // Test if the url is for a worker script (convention: workerXX.js), if so get its number
   const [_, workerId] = url.match(/\/worker(\d\d?).js$/) || [];
   if (workerId) {
      const workerScript = `
         let isInit = false;
         let functions = null;
         self.onmessage = async ({data}) => {
            if (isInit && data.functionName) {        // COMMON TASKS
               if (data.args) {
                  postMessage( functions[data.functionName](...data.args) );
               } else {
                  postMessage( functions[data.functionName]() );
               }
            } else {    // INSTANTIATE THE Wasm MODULE WITH ITS ENVIRONMENT

               try {
                  const byteSize = data.bufferSize * 2**16;
                  const id = new WebAssembly.Global({ value: 'i32', mutable: false }, ${workerId});
                  const max = new WebAssembly.Global({ value: 'i32', mutable: false }, data.threadNumber);
                  const size = new WebAssembly.Global({ value: 'i32', mutable: false }, byteSize);
                  data.imports.shared.size = size;
                  data.imports.thread = { id, max };
                  ({ exports: functions } = await WebAssembly.instantiate(data.module, data.imports));
                  isInit = true;
                  postMessage('OK');
               } catch(error) {
                  postMessage('FAILURE');
               }

            }
         };
      `;
      event.respondWith(new Response(workerScript, responseOptions));    // Returns a worker script
   } else {             // if other request, propagate the request outside the Service Worker
      event.respondWith(fetch(event.request));
   }
}
```

4.1.1.5. Thread setup - main program side

⚠️ Activated server pointing to the folder with **sw.js** required ⚠️

4.1.1.5.1. Explanations

In the main program we need to define a number of threads, and the size in pages of the *SharedArrayBuffer*, and our usual helpers to convert hexadecimal text to binary code :

```
const threadNumber = 2;
const bufferSize = 10;  // values in pages for the initial and maximum sizes for the Wasm memory
let memory;                    // will contained the Wasm memory encapsulating the SharedArrayBuffer

function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
   return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}

const wasmHeader = `00 61 73 6D 01 00 00 00`;
const wasmSource = `${wasmHeader}
            ....          // Wasm binary code as hexadecimal
`;
```

Then we need to register (activate) our **sw.js** service worker - so that can can dynamically forge worker scripts - as follows :

```
await navigator.serviceWorker.register('/sw.js', { scope: '/' });
```

After that we can initialize our threads, through an **initThreads** utility :

```
const threads = await initThreads(wasmSource, threadNumber, bufferSize);

async function initThreads(wasmSource, threadNumber, bufferSize) {
   // we only compile the Wasm binary code to a module, the instantiation is the responsibility of each worker
   const module = await WebAssembly.compile(wasmConvertDirtyHexToBinary(wasmSource));
   // we create the SharedArrayBuffer, hidden inside a Wasm Memory object, to be shared with each worker
   const memoryDescriptor = {
      initial: bufferSize          // value in pages : 1 page = 65 536 bytes
      maximum: bufferSize    // ⚠️  must be defined for shared memory
      shared: true                 // ⚠️  mandatory for shared memory => SharedArrayBuffer instead of ArrayBuffer
   };
   memory = new WebAssembly.Memory(memoryDescriptor);
   // we prepare the Wasm imports object, to be completed by each worker
   const imports = { shared: { memory } };
   // we create our threads via a small Thread class utility (see next page)
   const threads = new Threads(threadNumber);
   // we instantiate the compiled Wasm module in each worker
   // the run method basically does a postMessage on every worker with its first argument as data.
   // in this case, the code executed in each worker is the one below
   // the “INSTANTIATE THE Wasm MODULE WITH ITS ENVIRONMENT” comment (see previous page)
   const initStatuses = await threads.run({ module, imports, threadNumber, bufferSize });
   console.info(`Thread initialization: ${initStatuses}`);
   return threads;
}
```

Then we can ask the threads to perform tasks, and expect a notification when it's over :

```
const results = await threads.run({     // run the same task defined by the object message on all workers
   functionName: 'func',                // run the Wasm function named 'func' on each worker (splitted task)
   args: []                                    // arguments for the 'func' Wasm function, passed inside an array
});
console.log(results)          // results can be empty (the result being the processed SharedArrayBuffer)
```

The code above relies on a small **Threads** class :

```
class Threads {
   constructor(threadNumber) {
      this.workers = Array.from(Threads.create(threadNumber));     // See appendix A.0.2.8.
      this.workerResults = [];;           // where the results of the last task (run) are stored, ordered by worker id
   }

// Create a worker at each call < threadNumber, see Appendix A.0.8 for explanation
   static *create(threadNumber) {
      let i = 0;
      while (i < threadNumber) {
         yield new Worker(`worker${i + 1}.js`);
         i++;
      }
   }

// Post a payload on a worker and listen for a result, returned as a Promise.
   static runSingleTask(worker, payload) {
      return new Promise(resolve => {
         worker.onmessage = ({data}) => {
            worker.onmessage = null;  // to avoid side-effects
            resolve(data);
         };
         worker.postMessage(payload);
      })
   }

// Run a task (Wasm initialization, threaded Wasm function)
// Post a payload on every worker and store an array with the ordered results
   async run(payload={}) {
      this.workerResults = await Promise.all(
         this.workers.map(worker => Threads.runSingleTask(worker, payload))
      );
      return this.workerResults;          // even if the response is empty, this gives an end notification
   }
}
```

Note : static methods - no dependency on the class instance **this** *- are class properties, consequently called as* **Threads** *methods.*

4.1.1.5.2. Code summary

Open a page at your server root, declare a **binaryCode** variable with the hexadecimal binary code, and paste the code below in the console to run a task on multiple threads :

```
// Constants and Utilities
        const threadNumber = 2;
        const bufferSize = 10;
        let memory;

        function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
          return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
        }
        const wasmHeader = `00 61 73 6D 01 00 00 00`;
        const wasmSource = `${wasmHeader} ${binaryCode}`;

        class Threads {
          constructor(threadNumber) {
            this.workers = Array.from(Threads.create(threadNumber));     // See appendix
            this.workerResults = [];
          }
          static *create(threadNumber) {
            for (let i = 0; i < threadNumber; i++) { yield new Worker(`worker${i + 1}.js`); }
          }
          static runSingleTask(worker, payload) {
            return new Promise(resolve => {
              worker.onmessage = ({data}) => {
                worker.onmessage = null;  // to avoid side-effects
                resolve(data);
              };
              worker.postMessage(payload);
            })
          }
          async run(payload={}) {
            this.workerResults = await Promise.all(
              this.workers.map(worker => Threads.runSingleTask(worker, payload))
            );
            return this.workerResults;
          }
        }

        async function initThreads(wasmSource, threadNumber, bufferSize) {
          const module = await WebAssembly.compile(wasmConvertDirtyHexToBinary(wasmSource));
          const memoryDescriptor = { initial: bufferSize, maximum: bufferSize, shared: true };
          memory = new WebAssembly.Memory(memoryDescriptor);
          const imports = { shared: { memory } };
          const threads = new Threads(threadNumber);
          const initStatuses = await threads.run({ module, imports, threadNumber, bufferSize });
          console.info(`Thread initialization: ${initStatuses}`);
          return threads;
        }

// Threaded Wasm Setup
        await navigator.serviceWorker.register('/sw.js', { scope: '/' });
        const threads = await initThreads(wasmSource, threadNumber, bufferSize);

// Start a threaded Wasm function
        const results = await threads.run({ functionName: 'func', args: [] });
        console.log(results)            // results can be empty (the result being the processed SharedArrayBuffer)
        const arr = new Float32Array(memory.buffer);
        console.log('Processed Shared array viewed as a Float32Array:');
        console.log(arr);
```

4.1.2. Basic Multithreading

4.1.2.1. Assumptions

Multithreading introduced in this chapter is basic, in the sense that :

- each Wasm instance (one per worker) only modifies its own portion of the common *SharedArrayBuffer*. Say that the *SharedArrayBuffer* size is 800,000 bytes, and that Wasm instances read and write items inside as **i32** values. Consequently, this is virtually a 200,000 element array. Then, if we have 2 workers, we can assign the processing of the first 100,000 elements to **worker1**, and the remaining to **worker2**.
- each worker is expected to return a single result : no automatic restart in case of failure, so the whole task is lost.

Hence, the task organisation is so straightforward that there's no need to use specific operations designed to deal with concurrency (mutex/futex, **Atomics** global object, Wasm atomics operators).

The dummy threaded function introduced here - exported as '**func**' in each Wasm instance - simply initializes an *typed array* (our *SharedArrayBuffer*, filled by default with null values) with values corresponding to their indices. So it needn't arguments, and returns nothing (the processed array is the side effect).

4.1.2.2. Threaded Wasm module structure

The module imports :

- the memory object used as an array (based on a *SharedArrayBuffer*);
- 3 globals corresponding to the worker id, the number of workers and the size in bytes of the memory used as our array.

It defines 2 helpers functions :

- one function returning a memory limit for the processing (function with index 0):
  - the memory address (index) at which the loop starts if input is worker id;
  - the one at which the loop ends if input is worker id minus one.
- another's to process the *SharedArrayBuffer* part between the limits (start and end indices) (function with index 1 in the code at the next pages);

4.1.2.3. WAT Code

```
(module
    (type ( func (param i32) (result i32) ) )  ;; signature with index 0 - helper to determine an array boundary
    (type ( func (param i32) (param i32) ) )    ;; signature with index 1 - looping over an array part
    (type (func) )                              ;; signature with index 2 - “func” (exported function)
    (import "shared" "memory" (memory 10 10 shared) )    ;; array (memory as SharedArrayBuffer)
    (import "thread" "id" (global i32) )        ;; global with index 0 - workerId
    (import "thread" "max" (global i32) )       ;; global with index 1 - number of workers
    (import "shared" "size" (global i32) )      ;; global with index 2 - size of the memory (in bytes)
    ;; function with index 0 - determine the first or last byte address to be processed by the Wasm instance
    (func (type 0)              ;; function with index 0
        get_local 0             ;; function argument : if value = workerId, returns the last array address
                                ;; (excluded); if value = workerId - 1, returns the starting address
        f32.convert_i32_u       ;; every integer must be casted to float as decimals matter
        get_global 1            ;;  number of workers
        f32.convert_i32_u
        f32.div                 ;; workerId (or workerId - 1) / number of worker => array fraction
        get_global 2            ;; array size in bytes
        f32.convert_i32_u
        f32.mul                 ;; array fraction * array size => start or end index (address)
        i32.trunc_f32_u         ;; cast f32 to i32 (array index as a memory address)
    )
    ;; function with index 1 - looping over specific array part
    (func (type 1)      ;; called with minimum and maximum (process stopped right before) array indices
        block
            loop                        ;; br 0 ends here (below, address = index)
                get_local 0             ;; loop counter (starts at minimum index)
                get_local 0             ;; get the loop counter ..
                i32.const 4             ;;  .. but divide it by 4 to ..
                i32.div_u               ;;  … get number contiguous to the previous one
                f32.convert_i32_u
                f32.store
                get_local 0
                i32.const 4             ;; next value index  is 4 bytes further
                i32.add                 ;; new value for the loop counter
                tee_local 0             ;; save the value but keep it upon the stack
                get_local 1             ;; maximum array index
                i32.ge_u                ;; is loop counter greater or equal?
                br_if 1                 ;; exit the loop
                br 0                    ;; ...else repeat
            end
        end                             ;; br_if 1 ends here (exit)
    )
    (func (type 2)          ;; main function
        (local i32)         ;; first local (index 0) for storing starting index
        get_global 0        ;;  workerId
        call 0              ;; return last index (array processing ends here - excluded)
        set_local 0         ;; save it
        get_global 0
        i32.const 1
        i32.sub             ;; workerId - 1
        call 0              ;; return first index
        get_local 0         ;; last index
        call 1              ;; start the processing loop
    )
    (export "func" (func 2))
)
```

Note : to transpile bleeding-edge WAT code please see Appendix A.4.

4.1.2.4. Binary code / code summary

WAT token	i32 (value type)	f32.convert_i32_u	get_global	f32.mul	f32.div	f32.mul	i32.trunc_f32_u	i32.div_u	i32.ge_u
Binary token	*7F*	*B3*	*23*	*94*	*95*	*94*	*A9*	*6E*	*4F*

```
let binaryCode = `
01          <- Type list : Section list for defining types (parameters and results)
  0E(length) 03(number of defined types (number of types sections))
  60    01 7F(arguments)          01 7F (result)                          index-0 type
  60    02 7F 7F(arguments)       00(result)                              index-1 type
  60    00(arguments)             00(result)                              index-2 type

02          <- Import list : Section list for importing items from javascript
  3D(length) 04(number of imported items)
  06(length) 73(s) 68(h) 61(a) 72(r) 65(e) 64(d)   06(length) 6D(m) 65(e) 6D(m) 6F(o) 72(r) 79(y)
  02(import category : memory)      03(shared memory)      0A(initial size)     0A(maximum size)
  06(length) 74(t) 68(h) 72(r) 65(e) 61(a) 64(d)   02(length) 69(i) 64(d)     index-0 global
  03    (import category : global)  7F(type)  00(constant global)
  06(length) 74(t) 68(h) 72(r) 65(e) 61(a) 64(d)   03(length) 6D(m) 61(a) 78(x)     index-1 global
  03    (import category : global)  7F(type)  00(constant global)
  06(length) 73(s) 68(h) 61(a) 72(r) 65(e) 64(d)   04(length) 73(s) 69(i) 7A(z) 65(e)     index-2 global
  03    (import category : global)  7F(type)  00(constant global)

03          <- Function list : Section list for defining functions
  04(length) 03(function number) 00(1st function: signature index)  01(2nd function : same)  02(3rd function : same)

07          <- Export list : Section list for exporting stuffs to JS-land
  08(length) 01(number of exported functions)
    04(length) 66(f) 75(u) 6E(n) 63(c)   00(export category : function)  02(function index in the function list above)

0A          <- Code list : Section list for defining function bodies
  48(length in bytes) 03(number of code sections)
    0E(code section length) 00 (number of extra variables)                          1st code section
    20(get_local) 00        B3(cast to float) get the parameter at index 0 in the function signature as float
    23(get_global) 01       B3(cast to float) get the number of worker ("max" global)
    95(div)                 determine an array fraction
    23(get_global) 02       B3(cast to float) get the array size ("size" global)
    94(mul)                 A9(cast to integer)  fraction * size => get the start or end (excluded) address
    0B                      notify end of code block

    21(code section length) 00 (number of extra variables)                          2nd code section
      02(block) 40(returns nothing)      br 0 ends here
        03(loop) 40(returns nothing)
          20(get_local) 00      address of the value to be changed in the SharedArrayBuffer
          20(get_local) 00  41(const) 04  6E(div)   B3(cast to float)     value to be stored (typed array index)
          38(store) 00 00       store value above at the memory address 2 lines above
          20(get_local) 00 41(const) 04 6A(add) 22(tee_local) 00   increment and set loop counter (kept on the stack)
          20(get_local) 01 4F(greater or equal)                    is the array ending address reached?
          0D(br_if)  01         exit loop
          0C(br) 00             ...else repeat
        0B                      notify end of loop section
      0B                        notify end of block section - br_if 1 ends here
    0B                          notify end of code block

    15(code section length)                                                         3rd code section
    01 (number of extra variables) 01 7F                    one extra variable of type 4-byte integer
    23(get_global) 00       get the worker id
    10(call) 00             execute function with index 0 with worker id
    21(set_global) 00       save result in local 0
    23(get_global) 00 41(const) 01 6B(sub)      get the worker id minus one
    10(call) 00             execute function with index 0 with worker id minus one => min. array address
    20(get_local)  00       get previous result => max. array address
    10(call) 01             call function with index 1 (loop) with min. and max. addresses
    0B                      notify end of code block`;
```

Copy the code below into your console, then the one of chapter 4.1.5.2. and look at the last console message to check the processed *SharedArrayBuffer*, displayed as a Float32Array.

4.1.3. Standard Multithreading - atomics

🙏 Retweet pinned tweet on my 👉 twitter account as a token of appreciation (extra motivation to add new chapters)!

4.2. Vector processing (SIMD)

⚠️ Feature in development => Not for production. For tinkering, start the last chrome version from the command line like this : **google-chrome --js-flags="--experimental-wasm-simd"** ⚠️

4.2.1. Context

Even before the availability of multicore processors, CPU manufacturers and platform designers offered us the possibility to boost repetitive and numerical routines - like matrix multiplications or making loops smaller (ex: 200 iterations instead of 800 ones) - through *Single Instruction Multiple Data* (SIMD) technologies like MMX, SSE2 (Intel™) or 3DNow! (AMD™). Basically, provided some contiguous and properly aligned memory values, SIMD technologies typically allow to process 16 bytes (128 bits) of memory at once, as an example by summing 4 **i32** values with 4 other's via a single atomic (= single CPU cycle) instruction. Currently, the most popular technologies belong to the SSE family for Intel™/AMD™ and to the Neon's for ARM processors (smartphones/tablets), but this is an implementation detail as WebAssembly SIMD is focused on a common, minimum basis of 128-bit processing.

4.2.2. Concepts

The 128-bit memory slabs processed by SIMD operators are called vectors - whose type is **v128** (**7B** in binary). You can use them :

- with regular **get_local** and **set_local** instructions;
- with **v128.load** and **v128.store** instructions - behaving like **i32.load** and **i32.store** ones, respectively. However, as such instructions are designed to work on 128-bit-aligned memory values, you must specify 4 as the alignment parameter (chapter 2.2.3.5.3.)
- in internal function signatures, but you can't use **v128** values in interfaces - imported or exported stuffs. Hence, you have to read the results in the buffer of your Wasm memory, that must naturally be imported and/or exported.

v128 byte sequences can be viewed from our side as "shapeshifting" arrays, either as :

1. $[W_1, W_2]$ where W are represented by **i64/f64**;
2. $[X_1, X_2, X_3, X_4]$ where X are half-W (**i32/f32**).
   for instance W_1 bits are the concatenated ones of X_1 and X_2
3. $[Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8]$ where Y are half-X (*word* = 16 bits);
4. $[Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7, Z_8, Z_9, Z_{10}, Z_{11}, Z_{12}, Z_{13}, Z_{14}, Z_{15}, Z_{16}]$
   where Z are half-Y (byte).

Hence, an instruction must specify the shape of the v128 bit "array", using the concept of *lanes*, that is - for a given SIMD operation - the number of contiguous values individually processed at once :

1. 2 lanes if **f64/i64** values - **v64x2** structure;
2. 4 lanes if **f32/i32** values - **v32x4** structure;
3. 8 lanes if words - **v16x8** structure (items accessed as **i32**);
4. 16 lanes if bytes - **v8x16** structure (items accessed as **i32**).

Say that you've loaded such a 128-bit structure, starting at a given offset of your Wasm memory : you may need additional instructions before starting useful SIMD instructions like summing multiple values at once :

- you can prepare another **v128** value from a basic type through *splatting* instructions : if you pass a given value (**i32** automatically casted to bytes or words) to such an instruction, it will push upon the stack a **v128** value where each component has the same value. For instance **f32.const 2.2** followed by **f32x4.splat** returns a **v128** value consisting of elements [2.2, 2.2, 2.2, 2.2] if viewed as **f32 v32x4**;
- you can extract *lane content* through *extract_lane* instructions. As an example, with **v128** [42, 50, 33, 67] as the penultimate value upon the stack and 2 as the last one (index of the lane value to be extracted), **i32x4.extract_lane** returns 33.
- you can replace *lane content* through *replace_lane* instructions. For instance with **v128** [42, 50, 33, 67] as the antepenultimate value (1st argument), 0 as the penultimate one (2nd argument : index of the lane value to be replace) and 7 as the last one (3rd argument), **i32x4.replace_lane** returns **v128** [7, 50, 33, 67].
- you can re-organize and even make null some elements constituting a vector through *swizzling* instructions, whose 1st argument is the vector to be re-organized, and the 2nd one is a vector containing new indices for the modification. For instance, with **v128** [42, 50, 33, 67] as the penultimate value upon the stack (1st argument) and **v128** [3, 2, 1, 0] at the top of the stack (2nd one), a swizzling instruction would push **v128** [67, 33, 50, 42] upon the stack.
- you can extract some lanes from a vector and combine them to produce a new vector through *shuffling* instructions, whose operation is a bit more tricky to understand : the first two arguments are the **v128** vectors used as lane sources, and the 3rd one is an immediate argument - not upon the stack, but directly follows the instruction like *relative offset* and *alignment* for memory instructions - representing an index list for all the lanes (8 indices for 2 **v128** arguments of shape **v32x4** - 4 lanes each), allowing to shuffle them. The subtlety is that the index list concerns the first argument vector but if the index value is above the lane number (4 in the example below) : in that case it corresponds to the second argument vector, and its real value is found by subtracting the lane number.
  As an example for **v128** [1, 2, 3, 4] then **v128** [5, 6, 7, 8] upon the stack (4 lanes),
  v32x4.swizzle 0 4 1 5 would return [1, 5, 2, 6] (index list bytes in blue)

  Note: Shuffling and swizzling examples are only given for simplicity's sake as they only exist for **v8x16 v128** *at the moment, and not for* **v32x4**.

The following basic example consists in a SIMD basic **f32** multiplication, hopefully allowing you to grasp the concept :

- in the Wasm memory, the **v128** result will be stored in the 16 first bytes (4 **f32** values);
- the v128 inputs are stored at offsets 16 (**10**) and 32 (**20**). Hence, once our memory is viewed as a **Float32Array** *typed array*, one has :
    - before the multiplication :

      [**0**, 0, 0, 0, **1**, 2, 3, 4, **2**, 4 8, 16]

      byte addresses: **0 4 8 12 16 vector1 32 vector2**
    - after :

      [**2**, 8, 24, 64, **1**, 2, 3, 4, **2**, 4 8, 16]

      with 2 = 1 * 2, 8 = 2 * 4, 24 = 3 * 8 and 64 = 4 * 16

4.2.3. WAT code

```
(module
        (type                           ;; type  with index 0 in type list
                (func)                  ;; no params nor result, side-effects in the memory
        )                                       ;; … unsigned byte value at this offset (casted to i32)
        (memory 1 1)                    ;; 64 kB (1 page) memory, also limited to 1 page (cannot be grown).
        (func (type 0)                  ;; function with index 0 in function list and signature with index 0 in type list
                (local v128)            ;; extra local to store  the multiplication result
                i32.const 16            ;; load f32 values at the memory address 16 : vector 1
                v128.load
                i32.const 32            ;; load f32 values at the memory address 32  : vector 2
                v128.load
                f32x4.mul               ;; SIMD f32 multiplications (4 at once!) between vectors 1 and 2
                set_local 0             ;; save the result
                i32.const 0             ;; memory address at which the v128 result will be stored
                get_local 0             ;; push the argument (offset) upon the stack
                v128.store              ;; v128 saved
        )
        (export "mul" (func 0))
        (export "mem" (memory 0))
)
```

Notes :

- *you can transpile this only with the official tool (WIP) (check first the “simd” checkbox!): https://webassembly.github.io/wabt/demo/wat2wasm/*
- *the main function can be rewritten as follows - by setting first the memory offset to save the result - if we want to minimize the instructions :*

```
        (func (type 0)                  ;; function with index 0 in function list and signature with index 0 in type list
                i32.const 0             ;; stack (after the instruction): [..., 0]

                i32.const 16            ;; stack: [..., 0, 16]
                v128.load               ;; stack: [..., 0, vector1]
                i32.const 32            ;; stack: [..., 0, vector1, 32]
                v128.load               ;; stack: [..., 0, vector1, vector2]
                f32x4.mul               ;; stack: [..., 0, result]

                v128.store              ;; stack: […]       -       side-effect : result saved in Wasm memory
        )
```

4.2.4. Binary code / code summary

⚠️ SIMD binary instruction are encoded in 2 bytes instead of one : as hexadecimal values, the first one is always **FD** and the second one refer to the specific instruction => please check the instruction list here, with the byte instruction in the simdop column. Type **v128** as binary token **7B** ⚠️

```
(async () => {
function wasmConvertDirtyHexToBinary(wasmDirtyHexText) {
  return Uint8Array.from( wasmDirtyHexText.match(/[\dA-F]{2}/g).map( hex => parseInt(hex, 16) ) );
}
const wasmHeader = `00 61 73 6D 01 00 00 00`;
function convertIEEE754toHexa(value) {
   const arr = [null, null, null, null, null, null, null, null];
   const ab = new ArrayBuffer(8);        const dv = new DataView(ab); dv.setFloat64(0, value);
   for (let i = 0;  i < 8; i++) { arr[8 - i] = dv.getUint8(i).toString(16).padStart(2, '0').toUpperCase(); }
   return arr.join(' ');
}
const wasmSimdExample = `${wasmHeader}
01          <- Type list : Section list for defining types (parameters and results)
   04(length in bytes) 01(number of defined types : number of types sections)
   60       define a function type for type with index 0
   00       number of function arguments and types
   00       number and type of returned values

03          <- Function list : Section list for defining functions
   02(length in bytes) 01(number of defined functions)
   00       first not-imported function : index of the corresponding type in the type list above

05          <- Memory list : Section list for defining “slabs” of memory
   04(length in bytes) 01(number of defined memory slabs (always 1 for Wasm 1.0!))
   01(constant size memory) 01(initial size) 01(maximum size)

07          <- Export list : Section list for exporting stuffs to JS-land
   0D(length in bytes) 02(number of exported items)
      03(string length in bytes) 6D(m) 75(u) 6C(l)                                    ”mul”
      00    export category (0 for function, 1 for table, 2 for memory, 3 for global)
      00    function index  in the function list above
      03(string length in bytes) 6D(m) 65(e) 6D(m)                                    ”mem”
      02    export category (0 for function, 1 for table, 2 for memory, 3 for global)
      00    memory index  in the memory list above

0A          <- Code list : Section list for defining function bodies
   1E(length in bytes) 01(number of code sections)
      1C(length in byte of the code section below)
      01 (number of local variables other than arguments) 01 7B(one local vector)      local variable to store a vector
      41(const) 10                                                                     load 1st vector
      FD 00(load) 04(alignment) 00(relative offset)                                    sixteen-byte load with sixteen-byte alignment
      41(const) 20                                                                     load 2nd vector
      FD 00(load) 04(alignment) 00(relative offset)                                    sixteen-byte load with sixteen-byte alignment
      FD 9C(single precision float vector multiplication)                              perform multiplication between the 2 vectors
      21(set_local) 00                                                                 result saved into the extra local
      41(const) 00                                                                     memory address for storing the result
      20(get_local) 00                                                                 value to be saved (result)
      FD 01(store) 04(alignment) 00(relative offset)                                   result is saved
      0B(end of code block)`;
let { instance: { exports: { mul, mem } } } = await WebAssembly.instantiate(
   wasmConvertDirtyHexToBinary(wasmSimdExample)
);          const arr = new  Float32Array(mem.buffer);
Object.assign(arr, [0,0,0,0,1, 2, 3, 4, 2, 4, 8, 16]);
window.alert(`Wasm : initially, the f32 memory array is: [${arr.toString().substring(0, 32)}, …]`);
mul();
window.alert(`Wasm : Result (4 first values) is: [${arr.toString().substring(0, 32)}, …]`); })();
```

Appendix

A.0. Prerequisites

A.0.1. Hexadecimal byte

Example: to determine the value of byte **0100 0010**, one use power-of-two for its high bits :

2^{nd} bit => $2^1 = 2$; 7^{th} bit => $2^6 = 64$ whose summation yields 66

Consequently, **abcd efgh** in binary is equivalent to :

$a \cdot 2^7 + b \cdot 2^6 + c \cdot 2^5 + d \cdot 2^4 + e \cdot 2^3 + f \cdot 2^2 + g \cdot 2^1 + h = 2^4 \cdot (a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d) + e \cdot 2^3 + f \cdot 2^2 + g \cdot 2^1 + h \cdot 2^0 = 2^4 \cdot \mathbf{X} + \mathbf{Y}$

where **X** and **Y** are in the 0-15 range, called *hexadecimal*, and are simply expressed as a **XY** string when using **parseInt** with 16 as a second parameter : **parseInt('XY', 16) => value**

Correspondence : direct till 9, then **A** for 10, **B** for 11, **C** for 12, **D** for 13, E for 14 and **F** for 15

A value is given in the hexadecimal notation by prepending **XY** with **'0x'** (**0x42** => 66).

A.0.2. Javascript

The javascript used in this book is the new one, called ES6+. Here are the changes for the courageous readers who've "only" got a jQuery background! (exception: typed arrays)

A.0.2.1. Typed arrays

Typed arrays are like arrays (important: do not share all their methods!), but under the hood are similar to structures used in lower level languages like C/C++ : their items are contiguous in the memory, and are … typed. The memory a *typed array* operates on is called an **ArrayBuffer**, and is defined by calling this constructor with its size in bytes as an argument :

```
const ab = new ArrayBuffer(10);          // declare a 10 byte continuous memory slab
```

You're not forced to use typed arrays as you can directly operate on this memory using a **DataView**, which defines an interface (multiple interfaces may exist for a given *ArrayBuffer*) :

```
const v = new DataView(ab);
v.setUint8(0, 42);    // Uint8 = byte : put value 42 in the first byte of the ab ArrayBuffer
v.getUint8(9);        // get the value in the 10th byte of the  ab ArrayBuffer
```

DataViews allow to operate with arbitrary types. However, most of the time you store a specific type into your *ArrayBuffer*, and only use *typed arrays* as follows :

```
const bytes = new Uint8Array(ab);     // create a typed array of unsigned bytes based on the ab ArrayBuffer
```

Or you can simply pass array maximum length as an argument :

```
const bytes = new Uint8Array(10);     // same
const arrayBuffer = bytes.buffer;     // get the ArrayBuffer (similar to ab) the typed array operates on
```

⚠️ *for typed arrays, endianness depends on the platform. On the contrary it's by default big endian for DataViews : you can get a value formatted as little endian by setting to* **true** *the second parameter of a getter (false by default => big-endian) :* **v.setInt16(0, 15000); v.getInt16(0, true)** ⚠️

A.0.2.2. Variables

In addition to the **var** keyword to declare variables in the scope of its function, you have the **let** and **const** keywords, whose scope is the current block code (as in many other languages).
The difference between **let** and **const** is that the primitive value or reference stored inside a variable declared with **const** cannot be changed.

Note: only the reference cannot be changed with **const** *:*

```
const obj = { value: 42 };
obj  = { value: 43 };          // throw an error!
obj.value = 43;                // OK, the reference isn't changed
```

A.0.2.3. Destructuring

Say you have this nested object :

```
const obj = {
   obj2: {
      obj3: { value: 42 }
   }
};
```

Then to access a nested value :

```
const value = obj.obj2.obj3.value;
```
is equivalent to
```
const { obj2: { obj3: { value } } } = obj;
```

The advantage of this approach is that you can declare default values as follows :

```
const { obj2: { obj3: { value = 42  } } } = obj;
```

You can also rename a variable by declaring its new name after a column :

```
const { obj2: { obj3: { value: newValue = 42  } } } = obj;
// now your variable is newValue instead of value
```

Also, say you have this array :

```
const arr = [42, 7, 3, 4];
```

Then you can retrieve the first and second values and remaining one - **rest** array - like this :

```
const [ first, second,  ...rest ] = arr;
```

A.0.2.4. Spread (...)

Say that you have an array, whose values you want to use as arguments for a function. Instead of using the **apply** method of the function, you can proceed like this :

```
function func(a, b, c) {
   ...
};
const arr = [1, 2, 3];
```

```
func(...arr);          // the array is spread as arguments:
```

Hence, **func(...[1, 2, 3])** is equivalent to **func(1, 2, 3)**

A.0.2.5. Objects

If you have a **name** variable and want to create an object having a key with this value and whose key name is as the variable's, then ES6 allows you to do this :

```
const obj = { name };          is equivalent to          const obj = { name: name };
```

Also, you can directly generate keys inside an object based on a string variable, as follows :

```
const key = 'value';
const obj = { [key]: 42 };      // obj = { value: 42 }
```

A.0.2.6. Arrow functions

Arrow function is an alternative to the usual function notation :

```
function(x, y) {                                    (x, y) => {    // optional parentheses if only one argument ( x => {}
)
   ...                    is equivalent to              ...
}                                                   };
```

If your function explicitly returns something, you can use this notation :

```
function(x) {
   return y;.             is equivalent to          x => y;    // an object literal result must be between parenthesis
}
```

Consequently :

```
function(x) {
   return function(y) {
      return z;
   };                     is equivalent to          x => (y => z);          itself equivalent to          x => y => z;
}
```

Finally, you can use array destructuring inside arguments :

```
function(x) {
   const y = x.y;
   return y;.             is equivalent to          ({ y }) => y;          // (y is a field inside the argument)
}
```

Notes:

- *Do not use arrow function if debugging is critical, as they are anonymous in the stack trace!*
- **this** *and* **argument** *inside a arrow function are those of the external context, so useful for event handler or to avoid* **this** *binding!*

A.0.2.7. Promise - await/async

ES6 offers us a new way to control the instruction flow through *Promises*. A *Promise* is an object with a **then** method, that is executed when a result is available (the moment a callback is usually executed). In this case, one says that the promise is resolved. Otherwise, if an error occurs, one says that the Promise is rejected. In both cases the Promise is fulfilled (this step cannot be reversed), while before it was in a pending state. As an analogy - not from me - when you order something, you're not directly given your order (the result), but a purchase receipt (a Promise in a pending state) : when the order is ready, **then** the purchase receipt (Promise) allows you to be given the order (result).
You can build a Promise as follows, with the **Promise** constructor, having as an argument a two-parameters function (**(resolve, reject) => { ... }** for its shape) :

```
const arbitraryPromise = new Promise( (resolve, reject) => {
   try {
      ...
      resolve(result)          // if everything is okay, the Promise is resolved with a result (then method executed)
   } catch(error) {
      reject(error)            // else the Promise is rejected with an error
   }
});
```

The 2 functions - **resolve** and **reject** - are provided by the inner environnement of the Promise object, and are used to transfert the result (ex: AJAX data) and error, respectively.
Hence, **arbitraryPromise** is used like this :

```
function handleResult(result) { ... }
function handleError(error) { ... }
arbitraryPromise.then(handleResult, handleError);
```

Where **handleResult** is executed right after **resolve(result)**, or **handleError** right after **reject(error)**.
One can also use this notation :

```
arbitraryPromise
   .then(handleResult)
   .catch(handleError);
```

If **handleResult** returns a value, the **then** method consequently returns a new Promise immediately resolved with the returned value. Concretely :

```
function handleResult(result) {
   return 2 * value;
}
function handleFinalResult(result) {
   console.log(result);
}
const nextPromise = arbitraryPromise.then(handleResult);
nextPromise.then(handleFinalResult);          // Display the value multiplied by two
```

Hence, you can chain promises like this, with the fetch API that returns a Promise :

```
fetch(/* your url */)
   .then(response => response.json()) // return a Promise with the response data parsed as a JSON
   .then(json => { ... })          // process the JSON
```

In this book, one sometimes says that a value is “wrapped” inside a Promise to mean that the **then** method passes this value as an argument to the inner function (ex: **handleResult**).

If you want to quickly define a resolved Promise, you can also use this static method :

```
const immediatlyResolvedPromise = Promise.resolve(42);
immediatlyResolvedPromise.then(result => { … });          // then is directly executed
```

Note: same logic for Promise.reject().

If you wait for various task completions, you can use the **Promise.all** method like this:

```
const initPart1Promise = new Promise((resolve, reject) => { … resolve(result1);  });
const initPart2Promise = new Promise((resolve, reject) => { … resolve(result2); });
const initPromise = Promise.all( [initPart1Promise, initPart2Promise] );
// when all the Promise in the array are resolved, return a fullfilled Promise where resultList = [result1, result2]
initPromise.then(resultList => { … });
```

await/async allow you to deal with Promises in a more “synchronous” way :

```
arbitraryPromise
  .then(handleResult)
  .catch(handleError);
```

is equivalent to

```
try {
  const result = await arbitraryPromise;
  handleResult(result);
} catch(error) {
  handleError(error);
}
```

When you use the **await** operator, the execution is stopped until the **arbitraryPromise** is resolved. Then, the result wrapped inside the Promise is returned.

If you use this pattern inside a function, you must prepend it with the **async** operator, like this :

```
async function processStuffs(item) { … }
```

or

```
async (item) => { … }
```

A.0.2.8. Generator

A generator is a regular function where a * token is appended after the **function** one, like this:

```
function* createWorkers(threadNumber) {
  let i = 0;
  while (i < threadNumber) {
    yield new Worker(`worker${i + 1}.js`);
    i++;
  }
}
```

Such a function can simulate multiple “return” via the **yield** keyword (the execution is not over, simply paused at the **yield** statement). Hence, the first time it’s called it “returns” a worker generated from the **worker1.js** code on the server, then the second time a worker from **worker2.js**, etc. Generators are *iterables*, and as such can be used to generate arrays or using array methods like :

```
Array.from(createThreads(2));          // [ Worker for worker1.js, Worker for worker2.js ]
```

A.1. IEEE-754-2008

A.1.1. Principle

IEEE-754 is a standard used to deal with floating point value. This format is already used by javascript (not specific to Wasm), and is better explained by an example, as follows :

Given a 64-bit floating point value *V*, the 64th bit (the higher one) represents the *sign S bit* (high only if negative V), the next 11 ones are for the *exponent E* and the 52 remaining ones are for the normalized *mantissa M (0 ≤ M < 2)* with 1 subtracted if the exponent is not null :

V = + or - (1 + M) · 2^E for E > 0 or **V = + or - M · 2^E for E = 0** (denormal numbers)

In order to obtain both E and M as an example with V = 42.3, one must repeatedly divide the value by two and increase the exponent accordingly :

42.3 / 2 = 21.15 E : 1
21.15 / 2 = 10.575 E : 2
21.15 / 2 = 5.2875 E : 3
10.575 / 2 = 2.64375 E : 4
2.64375 / 2 = 1.321875 E : 5
M = 1.321875 and E > 0 => 1 subtracted from M: V = + (1 + 0.321875) · 2^5

Finally, one converts M in digits by repeatedly multiplying by 2 and keeping the value before the comma, and subtracting it from the value to obtain the one for the next step :

0.321875 x 2 = 0.64375 value : 0
0.64375 x 2 = 1.2875 value : 1 (1 before the comma, 1.2875 - 1 = 0.2875 for the next step)
0.2875 x 2 = 0.575 value : 0
0.575 x 2 = 1.15 value : 1 ...
0.15 x 2 = 0.3 value : 0
0.3 x 2 = 0.6 value : 0
0.6 x 2 = 1.2 value : 1 | pattern starts here
0.2 x 2 = 0.4 value : 0 |
0.4 x 2 = 0.8 value : 0 |
0.8 x 2 = 1.6 value : 1 | pattern ends here

0.6 x 2 …. you recognized a pattern here (see 4 lines aboves), repeated to the infinity!
The mantissa in binary consists of the column values, from the top, truncated to 52 :

V = 0 00000101 0101001001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 10
11-bit exponent 52-bit mantissa with one subtracted (because non-null exponent)

Note: for 32-bit floats, differences are the exponent bit length (8), the mantissa bit one (23) and that 127 (called "bias") must be subtracted from the exponent digit value to obtain the real exponent value.

A.1.2. Converting a value to hexadecimal for binary code

```
const convertIEEE754toHexa = (() => {
   const arr = [null, null, null, null, null, null, null, null];
   const ab = new ArrayBuffer(8);
   const dv = new DataView(ab);
   return function(value) {
      dv.setFloat64(0, value);
      for (let i = 0;  i < 8; i++) {
         arr[8 - i] = dv.getUint8(i).toString(16).padStart(2, '0').toUpperCase();
      }
      return arr.join(' ');          // example: '00 00 00 00 00 00 00 40' if value equals 2
   };
})();
```

A.2. LEB128 compression

A.2.1. Example

The encoding of “big” numbers like the memory maximum value in chapter 2.2.3.1. (1000 becomes **E8 07**) explicitly uses the LEB128 compression (it is always used, but for small numbers the result is as if there were no compression, so no errors if you simply use binary values) :

1. *The formula is : 128 * higherByte + (lowerByte - 128)*
   *= 128 * **07** + (**E8** - 128)* *(remember the little-endian convention)*
   *= 128 * **7** + (**16 * 14 + 8** - 128) = 1000*
2. *Procedure to find **E8 07** : 1000 is equivalent, in bits, to **00000011 11101000***
3. *The digits must be splitted into 7-bit groups :* ***0000111*** ***1101000***
4. *Each group has to be prepended with a high bit but if the next, truncated value is null and if the 7th digit is null (termination condition):*

   ...0000 *(truncated value, omitted)* ***00000111*** *(both truncated value & 7th digit are null)* ***11101000*** *(7th digit not null)*

5. *In hexadecimal :* ***07 E8*** *, so **E8 07** in little-endian notation*

More precisely, as the value is signed, one defines the 7th digit in each “7-byte group” as a *sign bit*. In this case, the termination condition, underlined in step 5, can also be expressed as follows for a negative value :

- The next, truncated value is -1 : indeed, if the value is negative (e.g. -1000), the truncated bits are all high for a group with no more information instead of being null : indeed a group filled with 1 corresponds in *two’s complement* notation to -1 (11...111 => bit inversion => 000...000 => +1 => 00….001 but in negative, -1).
- In the current group, a negative value corresponds to a high sign bit.

Hence, the termination condition equivalence to :

“**next, truncated value is null and the sign bit is null**”

is “**next, truncated value is -1 and the sign bit is not null**”

A.2.2. Encoder

You can find below a LEB128 encoder to hexadecimal strings :

```
const encodeLeb = ( () => {
   const mask7Bits = 0b01111111;       // to build 7-bit groups (binary values are prepended with 0b in JS)
   const signBit = 0b01000000;         // 7th bit is defined as the sign bit
   const c = 0b10000000;               // 128
   let i, digitGroups;
   let more;
   const encode = (value) => {
      more = true;
      i = 0;
      digitGroups = [null, null, null, null, null, null, null, null, null, null];          // max 10 7-bit groups (i64/f64)
      while (more) {
         digitGroups[i] = value & mask7Bits;       // select current group of 7 bits
         value >>= 7;                              // value replaced by next, truncated value (sign conservation)
         if (  ( !value && !(digitGroups[i] & signBit) )          // positive and …
           || ( value == -1 && (digitGroups[i] & signBit) )      // … negative termination conditions (see last page)
         ) {
            more = false;                  // stop the encoding for the next, truncated value (no more information)
         } else {
            digitGroups[i] |= c;           // binary OR set on high the 8th bit
         }
         digitGroups[i] = digitGroups[i].toString(16).padStart(2, '0').toUpperCase();
         i++;
      }
      return digitGroups.filter(group => group !== null).join(' ');
   };
   return encode;   // hexadecimal string already ordered as little-endian
} )();
```

A.3. UTF-8

A.3.1. Principle

Every character of a given language is encoded by up to 4 bytes. How does the encoder infers character byte length ? As follows :

- 8th bit is low (ex: **0111 1111**) => 1-byte UTF-8 character (ex: latin characters);
- the X first bits are high and the (X+1)th is low => X-byte UTF-8 character (X != 1)
- Bytes following the first one must start by a high then a low bit.
  Example for a 2-byte UTF-8 character (X=2) : **1101 1111** (byte 1) **1011 1111** (byte 2)

A.3.2. Symbols and letters

	a	b	c	d	e	f	g	h	i	j	k	l	m
lower case	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**6A**	**6B**	**6C**	**6D**
upper case	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**4A**	**4B**	**4C**	**4D**

	n	o	p	q	r	s	t	u	v	w	x	y	z
lower case	**6E**	**6F**	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**	**7A**
upper case	**4E**	**4F**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**

space	!	"	#	$	%	&	'	(	)
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**

*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9
2A	**2B**	**2C**	**2D**	**2E**	**2F**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**

:	;	<	=	>	?	@	[	\	]	^	_	`
3A	**3B**	**3C**	**3D**	**3E**	**3F**	**40**	**5B**	**5C**	**5D**	**5E**	**5F**	**60**

A.4. Transpiling bleeding-edge WAT Code to Wasm binary

As most online services - like *WebAssembly Studio* - do not currently transpile bleeding-edge WAT code, we must use the official method thanks to NodeJs :

- create a folder;
- in a terminal and in this folder : **npm init -y**
- install the ‘wabt’ package : **npm i -S wabt**
- create a **server.js** file with the following content :

```
const { readFileSync, writeFileSync } = require('fs');
const wabt = require('wabt')();
const path = require('path');

const inputWat = 'main.wat';
const outputWasm = 'main.wasm';

const wasmModule = wabt.parseWat(inputWat, readFileSync(inputWat, 'utf8'));
const { buffer } = wasmModule.toBinary({});
writeFileSync(outputWasm, Buffer.from(buffer));
```

- create a **main.wat** file and copy inside your WAT Code;
- run **npm start** to create or update a **main.wasm** file containing the binary code.

You can then inspect the binary file with a hexadecimal editor, like **hexdump** on Linux :

```
hexdump -e '16/1 "%02x " "\n"' main.wasm | tr '[:lower:]' '[:upper:]'
```

(credits)